Prevenção Automática de Ataques de Não-Terminação

Raphael Ernani Rodrigues and Fernando Magno Quintão Pereira

Departamento de Ciência da Computação – UFMG – Brasil
{raphael,fernando}@dcc.ufmg.br

Resumo Dizemos que um programa é vulnerável a um ataque de não terminação quando um adversário pode lhe fornecer valores de entrada que façam algum de seus laços iterar para sempre. A prevenção de ataques desse tipo é difícil, pois eles não se originam de bugs que infringem a semântica da linguagem em que o programa foi feito. Ao contrário, essas vulnerabilidades têm origem na aritmética modular inteira de linguagens como C, C++ e Java, a qual possui semântica bem definida. Neste artigo nós apresentamos uma ferramenta que detecta tais problemas, e que saneia código vulnerável. A detecção da vulnerabilidade dá-se via uma análise de fluxo de informação; a sua cura decorre de guardas que nosso compilador insere no código vulnerável. Nós implementamos esse arcabouço em LLVM, um compilador de qualidade industrial, e testa-mo-no em um conjunto de programas que compraz mais de 2.5 milhões de linhas de código escrito em C. Descobrimos que, em média, caminhos em que informação perigosa trafega são pequenos, sendo compostos por não mais que 10 instruções assembly. A instrumentação que inserimos para prevenir ataques de não terminação aumenta o tamanho do programa saneado em cerca de 5% em média, e torna-os menos que 2% mais lentos.

Resumo We say that a program is vulnerable to a non-termination attack if (i) it contains a loop that is bounded by values coming from public inputs, and (ii) an adversary can manipulate these values to force this loop to iterate forever. Preventing this kind of attack is difficult because they do not originate from bugs that break the semantics of the programming language, such as buffer overflows. Instead, they usually are made possible by the wrapping integer arithmetics used by C, C++ and Java-like languages, which have well-defined semantics. In this paper we present the diagnosis and the cure for this type of attack. Firstly, we describe a tainted-flow analysis that detects non-termination vulnerabilities. Secondly, we provide a compiler transformation that inserts arithmetic guards on loop conditions that may not terminate due to integer overflows. We have implemented our framework in the LLVM compiler, and have tested it on a benchmark suite containing over 2.5 million lines of C code. We have found out that the typical path from inputs to loop conditions is, on the average, less than 10 instructions long. Our instrumentation that prevents this kind of attacks adds less than 5% extra code on the secured program. The final, protected code, is, on the average, less than 2% slower than the original, unprotected program.

1 Introdução

Um ataque de Negação de Serviços (*Denial-of-Service – DoS*) consiste em sobrecarregar um servidor com uma quantidade de falsas requisições grande o suficiente para lhe comprometer a capacidade de atender contatos legítimos. Existem hoje diversas maneiras diferentes de detectar e reduzir a efetividade desse tipo de ataque [13]. Neste artigo, contudo, descreveremos uma forma de negação de serviço que é de difícil detecção: *os ataques de não-terminação*. Um adversário realiza um ataque desse tipo fornecendo ao programa alvo entradas cuidadosamente produzidas para forçar iterações eternas sobre um laço vulnerável. Um ataque de não terminação demanda conhecimento do código fonte do sistema a ser abordado. Não obstante tal limitação, esse tipo de ataque pode ser muito efetivo, pois bastam algumas requisições para comprometer o sistema alvo. Uma vez que essa quantidade de incursões é pequena, os métodos tradicionais de detecção de negação de serviço não podem ser usados para reconhecer ataques de não-terminação. Além disso, dada a vasta quantidade de código aberto usado nos mais diversos ramos da indústria de *software*, usuários maliciosos têm a sua disposição um vasto campo de ação.

A detecção de código vulnerável a ataques de não-terminação é difícil. Tal dificuldade existe, sobretudo, porque esse tipo de ataque não decorre de deficiências de tipagem fraca, normalmente presentes em programas escritos em C ou C++. Programas escritos em linguagens fortemente tipadas, como Java, por exemplo, também apresentam a principal fonte de vulnerabilidades a ataques de não terminação: a *aritmética modular inteira*. Em outras palavras, uma operação como `a + 1`, em Java, C ou C++, pode resultar em um valor menor que aquele inicialmente encontrado na variável `a`. Esse fenômeno ocorrerá quando a variável `a` guardar o maior inteiro presente em cada uma dessas linguagens. Nesse caso, ao fim da operação, `a + 1` representará o menor inteiro possível em complemento de dois. Em outras palavras, um laço como `for (i = 0; i <= N; i++)` nunca terminará se `N` for `MAX_INT`, o maior inteiro da linguagem.

Este artigo traz duas contribuições relacionadas a ataques de não-terminação. Em primeiro lugar, ele descreve uma técnica que descobre vulnerabilidades relacionadas a esse tipo de ataque. Em segundo lugar, o artigo mostra como código pode ser protegido contra tais ataques. A nossa técnica de detecção de vulnerabilidades é baseada em análise de fluxo contaminado. Tal análise é parte do arcabouço teórico de rastreamento de informação inicialmente proposto por Denning e Denning nos anos setenta [6]. Um ataque de fluxo contaminado pode ser efetivo somente em programas cujas operações críticas dependam de dados de entrada. Em nosso contexto, uma operação crítica é o teste de controle de laço. Em conjunto com o algoritmo de detecção de vulnerabilidades, nós propomos também uma técnica para sanear programas contra ataques de não-terminação. Nossa estratégia consiste na inserção de verificações sobre as operações aritméticas realizadas pelo programa alvo. Essas verificações ocorrem durante a execução do programa, e invocam código de segurança sempre que estouros de precisão em variáveis inteira são percebidos. Nós instrumentamos somente código que controla o número de iterações em laços. Consequentemente, o arcabouço que

propomos incorre em uma perda de desempenho muito pequena, e, em nossa opinião, completamente justificável em decurso do benefício que assegura.

Nós implementamos todas as idéias que discutimos neste artigo em LLVM, um compilador de qualidade industrial [11]. Na seção 4 descreveremos uma série de experimentos que validam nossa análise. Ao analisar os programas presentes na coleção SPEC CPU 2006, fomos capazes de descobrir XX laços que são influenciados por dados provenientes de entradas públicas, isto é, que podem ser manipuladas por um adversário. Dentre esses laços, pelo menos XX estão sujeitos à não terminação. Para obter esse número, utilizamos um padrão muito simples: procuramos por laços cujo teste de parada é do tipo `i <= N`, sendo `N` dependente da entrada. Uma vez que nos atemos a esse tipo de condição de parada, especulamos que a quantidade de laços vulneráveis presente nos programas de SPEC CPU 2006 seja bem maior que o valor que apuramos. A nossa instrumentação – usada para impedir os ataques de não-terminação – mostra-se extremamente eficiente. Os testes que inserimos antes de cada operação aritmética que pode levar a um ataque de não terminação custa-nos uma perda de desempenho de menos que XX%.

2 Ataques de Não-Terminação

De acordo com Appel e Palsberg [1, p.376], um laço natural é um conjunto de nodos S do grafo de fluxo de controle de um programa (CFG), includindo um nodo cabeçalho H, com as seguintes três propriedades:

- a partir de qualquer nodo em S existe um caminho que chega a H;
- existe um caminho de H até qualquer nodo que faz parte de S;
- não existe caminho que não contenha H de qualquer nodo fora de S para qualquer nodo em S.

A *condição de parada* de um laço é uma expressão booleana $E = f(e_1, e_2, \ldots, e_n)$, sendo cada $e_j, 1 \leq j \leq n$ um valor que contribui para a computação de E. Seja P um programa que possui um laço L limitado por uma condição de parada $E = f(e_1, e_2, \ldots, e_n)$. Dizemos que P é vulnerável a um ataque de não terminação quando as duas condições abaixo são verdadeiras sobre ele:

1. existe um subconjunto $E' \subseteq \{e_1, e_2, \ldots, e_n\}$ de valores que dependem de um conjunto $I = \{i_1, i_2, \ldots, i_m\}$ de dados lidos a partir da entrada do programa.
2. existe uma atribuição de valores $i_1 \mapsto v_1, i_2 \mapsto v_2, \ldots, i_m \mapsto v_m$ que, ao influenciar E', faz com que o laço L não termine.

Note que a nossa definição de vulnerabilidade de não-terminação requer a noção de dependência de dados. Se um programa P possui uma instrução que usa a variável u e define a variável v, então v *depende* de u. Dependências de dados são transitivas, e embora possam ser cíclicas, não são necessariamente comutativas.

Ilustraremos ataques de não terminação via o programa mostrado na Figura 1(a). Esse programa calcula o fatorial de um número inteiro na linguagem

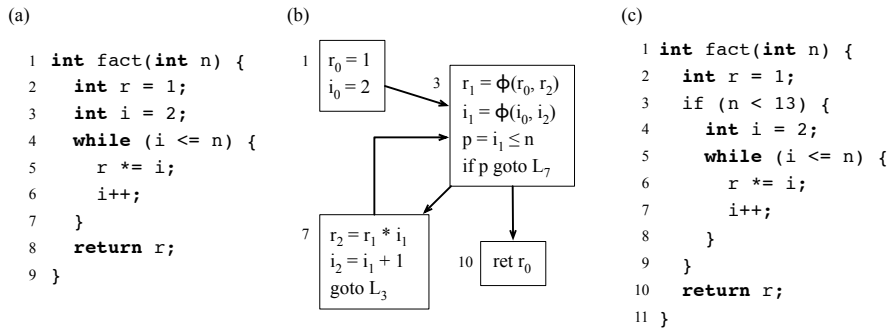

Figura 1. (a) Uma função em C, que calcula o fatorial de um número inteiro. (b) O CFG da função `fact`. (c) Exemplo de laço cuja condição de parada depende de valores de entrada mas que sempre termina.

C. O padrão que rege essa linguagem de programação não determina o tamanho do tipo `int`. Essa informação depende da implementação do compilador usado. Entretanto, é usual que inteiros sejam representados como números de 32 bits em complemento de dois. Nesse 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 terminará. A não-terminação ocorre porque quando `i` finalmente chega a MAX_INT, a soma $i + 1$ nos retorna o menor inteiro possível, isto é, -2^{31}.

O programa da figura 1(a) é vulnerável a ataques de não-terminação. Para explicitar tal fato, a figura 1(b) mostra o grafo de fluxo de controle do programa. Esse CFG está convertido para o formato de atribuição estática única (SSA) [5]. Usaremos essa representação de programas porque ela facilita a nossa análise de dependência de dados. Os blocos básicos que começam nos rótulos 3 e 7 formam um laço natural, segundo a definição de Appel e Palsberg. Esse laço é controlado pela condição de parada $i_1 \leq n$. A variável n, o limite do laço, é dependente da entrada. Existe um valor de n, a saber MAX_INT, que força o laço a não-terminar.

3 Detecção e Prevenção de Não-Terminações

Nesta seção descreveremos nossa técnica para detectar vulnerabilidades de não-terminação. Esse algoritmo de detecção fornece os subsídios necessários a uma segunda técnica que introduzimos neste artigo: o *saneamento de laços*.

3.1 Detecção Automática de Não-Terminação

Dizemos que um laço é *alcançável* quando as condições que o controlam usam valores que dependem de dados de entrada do programa. Note que um laço

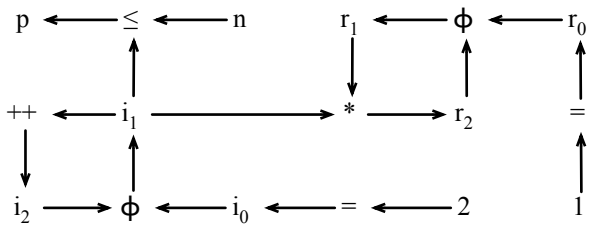

Figura 2. (a) Grafo de dependências da função `fact`, construído a partir do CFG visto na figura 1(b).

alcançável não é necessariamente vulnerável. A título de exemplo, o programa da Figura 1(c), uma sutil alteração da função `fact` inicialmente vista na figura 1(a), termina para qualquer entrada. A fim de determinar quais laços são alcançáveis, nós utilizamos o *grafo de dependências de dados*. Esse grafo é definido da seguinte forma: para cada variável v no programa, nós criamos um nodo n_v, e para cada instrução i no programa, nós criamos um nodo n_i. Para cada instrução $i : v = f(\ldots, u, \ldots)$ que define uma variável v e usa uma variável u nós criamos duas areastas: $n_u \leftarrow n_i$ e $n_i \leftarrow n_v$. O grafo de dependências que extraímos a partir d CFG visto na figura 1(b) é mostrado na figura 2.

Um caminho entre uma entrada do programa, e o predicado que controla o laço é uma condição necessária para um ataque de não-terminação. O grafo de dependências de nosso exemplo apresenta tal condição: existe um caminho que une o nodo correspondente ao parâmetro `n`, uma entrada, ao nodo que corresponde a `p`, o predicado de controle do laço. Esse tipo de caminho, uma vez contruído o grafo, pode ser encontrado em tempo linear no número de arestas do grafo - normalmente proporcional ao número de nodos - via a simples busca em profundidade ou largura.

Diremos que um laço é *vulnerável* quando ele é alcançável, e, além disso, sua condição de parada é dependente de alguma operação *cíclica* passível de estouro de precisão. Seguindo a definição de laços de Appel e Palsberg, uma operação cíclica é qualquer instrução que ocorre no corpo S do laço. Por exemplo, no CFG da figura 1(b), as instruções $i_2 = i_1 + 1$ e $r_2 = r_1 \times i_1$ são cíclicas. O laço daquele exemplo encaixa-se em nossa definição de vulnerabilidade, pois sua condição de parada é alcançável a partir da entrada, e depende de uma instrução cíclica passível de estouro de precisão: $i_2 = i_1 + 1$.

A nossa definição de vulnerabilidade inclui muitos laços que não são, de fato, vulneráveis, tais como aquele visto na figura 1(c). Seria possível utilizar técnicas computacionalmente intensivas, tais como algoritmos de satisfabilidade, para refinar a nossa definição, eliminando alguns desses falsos positivos. Tal abordagem já foi utilizada em trabalhos anteriores ao nosso [2,4,9,15,17]. Por outro lado, os próprios autores desses trabalhos reportam que dificilmente suas

técnicas poderiam lidar com programas muito grandes. Nós optamos por usar uma definição mais conservadora de laço vulnerável para termos uma ferramenta prática. Nós sanearemos todo laço considerado perigoso, inclusive aqueles que, devido à nossa definição liberal de vulnerabilidade, de fato não o são. Ainda assim, conforme mostraremos na seção 4, o impacto dessa instrumentação é negligível.

3.2 Saneamento de Laços

Uma vez encontrado um caminho vulnerável, passamos à fase de saneamento de laços. Um laço pode ser saneado via a inserção de testes que detectam e tratam a ocorrência de estouros de precisão inteira. Nós inserimos tais testes sobre as operações aritméticas *internas* que controlam a condição de parada do laço. Se S é o conjunto de blocos básicos que fazem parte do laço, uma operação é dita interna quando ela ocorre nalgum bloco em S. Continuando com o nosso exemplo, o laço alvo possui dois blocos básicos: o primeiro começa no rótulo três, e o segundo começa no rótulo sete. O laço possui duas operações aritméticas internas, todas ocorrendo no segundo bloco básico. Dentre essas operações, aquela no rótulo sete é inofensiva: ela define a variável r_2, que não participa da condição de parada do laço. Por outro lado, a operação no rótulo oito, que define a variável i_2, é usada no cálculo daquela condição, e precisa ser instrumentada.

Novamente, o grafo de dependências ajuda-nos a encontrar quais operações precisam ser instrumentadas para sanear um laço controlado por um predicado p. Nesse caso, usamos o seguinte critério para determinar se uma operação $i : v = f(v_1, \ldots, v_n)$ precisa ser instrumentada:

- existe um caminho no nodo n_i até o nodo n_p.
- O nodo n_i encontra-se em um ciclo.

A título de exemplo, a operação de incremento ++ no grafo de dependências visto na figura 2 precisa ser instrumentada. Em primeiro lugar, porque essa operação encontra-se em um ciclo. Em segundo lugar, porque existe um caminho do nodo n_{++} até o nodo n_p.

Instrumentação de Saneamento. A fim de garantir que estouros de precisão inteiros não venham a causar a não-terminação de laços, nós inserimos testes no código binário do programa alvo. O código que constitui cada um desses testes é formado por uma guarda, mas um tratador de eventos. Nossas guardas usam as condições mostradas na figura 3 para verificar a ocorrência de estouros de precisão. Atualmente instrumentamos quatro tipos diferentes de instrução: 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. Para ilustrar esse ponto, mostramos, na figura 4, o código necessário para instrumentar uma soma com sinal de duas variáveis. Essa figura mostra código no formato intermediário usado por LLVM,

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

Figura 3. Overflow checks. 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.

o compilador que utilizamos para implementar as idéias descritas neste artigo. Omitimos, nesse exemplo, o código do tratador de evento de estouro, pois ele simplesmente chamada 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, os experimentos que mostraremos na seção 4 indicam que somente uma parcela muito pequena das instruções do programa alvo precisam ser guardadas. Consequentemente, o custo, em termos de crescimento de código e perda de desempenho, é negligível.

4 Resultados Experimentais

Nós implementamos as técnicas descritas neste artigo em LLVM versão X.X. Nossa implementação foi testada em uma máquina Intel XX, com XX Gigabtyes de RAM, e XX GHz de Clock. Executamos nossa análise com sucesso sobre o arcabouço de testes de LLVM, um conjunto de programas com mais de 4.3 milhões de linhas de código C. No restante desta seção mostraremos somente resultados obtidos sobre os programas de SPEC CPU 2006.

Definição de Entrada de Dados: nos experimentos apresentados nesta seção, consideraremos como entrada de dados os seguintes valores:

– os argumentos do método `main`;

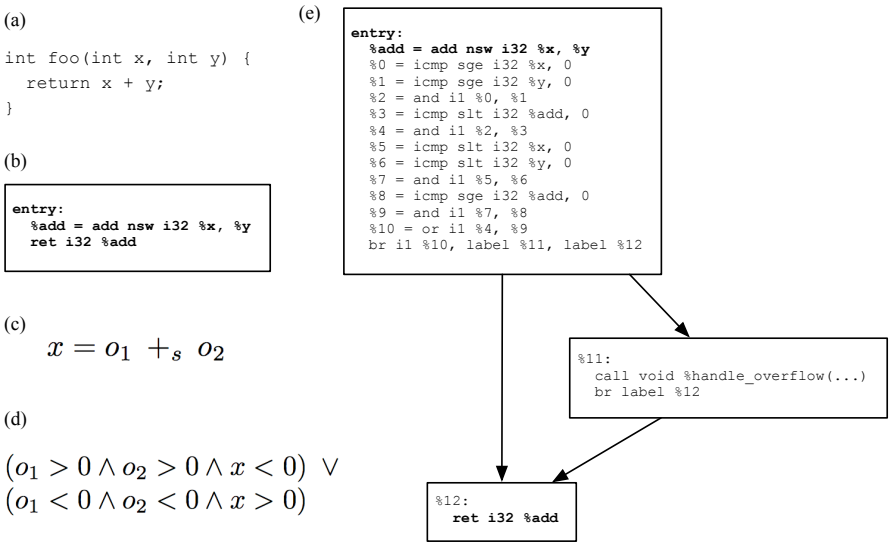

Figura 4. (a)

– o resultado retornado por funções *externas*;
– ponteiros passados como argumento de funções *externas*.

As funções externas são a união dos seguintes três conjuntos:

– funções que não foram declaradas em nenhum dos arquivos que compõem o programa compilado;
– funções sem corpo;
– funções que podem ser chamadas via um ponteiro de funções.

Laços Alcançáveis e Vulneráveis. A figura 5 mostra a quantidade de laços alcançáveis e vulneráveis que encontramos por programa. Estamos considerando, nesse caso, somente dependências de dados que não envolvam memória. Em outras palavras, todos os nodos do grafo de dependência são valores que podem residir em registradores. Essa versão de nosso detector não é segura: podemos deixar de marcar alguns laços vulneráveis como tal. Por outro lado, ela nos dá um número mínimo de laços que precisaríamos marcar, de acordo com a definição de laços alcançáveis da seção 3.1. Vemos, pela figura, que entre 1.3% e 19.3% de todos os laços do programa são alcançáveis. Aproximadamente a metade dos laços alcançáveis é vulnerável. Os laços que não são considerados vulneráveis em geral usam condições de paradas que comparam valores via igualdade, por exemplo, `while(a[i] != '\0') {i++;}`.

Benchmark	Num. Laços	Alcançáveis	Vulneráveis	Caminhos
433.milc	326	8	4	3
444.namd	508	7	1	2
447.dealII	4657	540	225	3
450.soplex	547	21	12	3
470.lbm	17	0	0	-
401.bzip2	205	1	1	3
403.gcc	3853	109	59	4
429.mcf	38	2	0	2
445.gobmk	1073	37	19	4
456.hmmer	687	133	48	2
458.sjeng	235	9	3	4
462.libquantum	67	2	1	4
464.h264ref	1586	27	22	15
473.astar	86	7	7	18

Figura 5. Informações estáticas inferidas pela análise de não-terminação. **Num. laços**: número de laços no programa. **Alcançáveis**: quantidade de laços que são dependentes de dados produzidos a partir de canais de entrada. **Vulneráveis**: número de laços que preenchem nossos requisitos de vulnerabilidade. **Caminhos**: tamanho médio do menor caminho de dependência de dados da entrada até a operação de controle do laço.

5 Trabalhos Relacionados

Este trabalho aborda temas relacionados a diferentes áreas da análise estática e dinâmica de programas, a saber: teoria de fluxo de informação, detecção de estouros de precisão inteira e análise de não-terminação. Além disso, este trabalho utiliza o conceito de *grafos de dependências*, inicialmente proposto por Ferrante *et al.* [8]. Em nosso caso, o grafo de dependência dá-nos a estrutura de dados básica sobre a qual caminhos que levam à não-terminação podem ser encontrados. Esses grafos, contudo, historicamente vêm se prestando a muitos outros propósitos, como escalonamento de instruções, detecção de condições de corrida e propagação de constantes, por exemplo.

Neste artigo usamos o grafo de dependências para rastrear o fluxo de informação contaminada. O rastreamento de fluxo de informação é uma grande sub-área dentro do campo de análise estática de programas [6]. Existem duas formas principais de rastrear a informação. Pode-se traçar o fluxo de dados a partir de operações sigilosas até entradas que um adversário pode ler. Esse modo de rastreamento é popularmente conhecido como detecção de vazamento de segredos [10]. E, no sentido inverso, pode-se traçar o fluxo de informação de entradas que um adversário pode manipular até operações críticas dentro do programa [16]. Essa categoria inclui nosso trabalho, além de diversos outros tipos de

Benchmark	Num. Instruções	Num. Arit.	Instrumentação	Crescimento
433.milc	22615	1165	7	0,50%
444.namd	65045	3136	1	0,03%
447.dealII	415756	14948	279	0,58%
450.soplex	60772	1815	19	0,46%
470.lbm	3614	1130	0	0,00%
401.bzip2	16359	1688	1	0,10%
403.gcc	792039	14853	92	0,14%
429.mcf	2541	158	0	0,00%
445.gobmk	130183	12158	54	0,63%
456.hmmer	61908	3585	86	2,00%
458.sjeng	27708	2438	20	1,10%
462.libquantum	5848	593	6	1,04%
464.h264ref	131905	13413	248	2,28%
473.astar	8261	639	9	1,73%

Figura 6. Impacto da instrumentação no código saneado. **Num. Instruções**: número de instruções no benchmark. **Num. Arit**: número de instruções que podem causar estouro de precisão inteira. **Instrumentação**: quantidade de testes inseridos para sanear o programa. **Crescimento**: razão entre o tamanho do programa instrumentado e o tamanho do programa original.

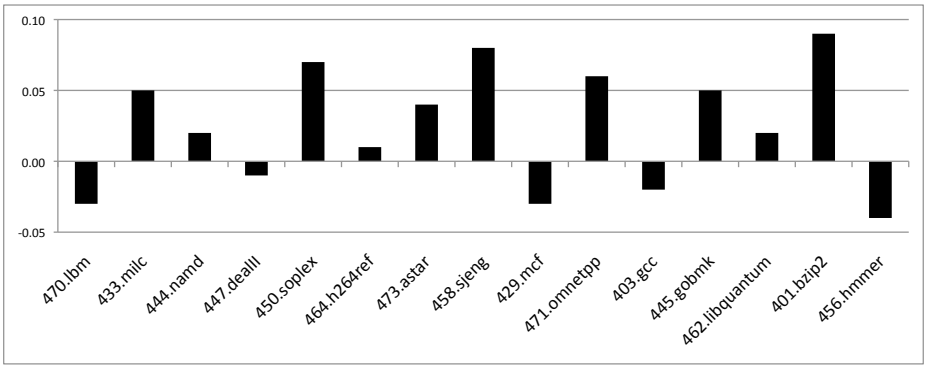

Figura 7. Variação no tempo de execução dos benchmarks devido à instrumentação de saneamento.

vulnerabilidades, tais como *Injeção de Código SQL* [18], *Injeção de Scripts* [14] e *Ataques de Estouro de Buffer* [12].

Nós instrumentamos código considerado vulnerável para detectar estouros de precisão que podem levar à não-terminação. Esses mesmos testes já foram usados com vários outros objetivos em trabalhos anteriores. O mais importante

trabalho nessa área deve-se, provavelmente, a Brumley *et al.* [3]. O grupo de David Brumley desenvolveu uma ferramenta, RICH, que instrumenta cada operação aritmética passível de estouro de precisão inteira em um programa C. A principal conclusão daquele trabalho foi que esse tipo de instrumentação não compromete sobremaneira o desempenho do programa modificado. RICH, por exemplo, aumenta o tempo de execução dos programas instrumentados em menos que 6% em média. Outro trabalho importante nesse campo foi publicado por Dietz *et al.* [7]. Esse grupo implementou IOC, uma ferramenta que, assim como RICH, detecta a ocorrência de estouros de precisão em operações aritméticas inteiras. Porém, ao contrário de Brumley *et al.*, Dietz *et al.* usaram sua ferramenta para desenvolver um amplo estudo sobre a ocorrência de estouros em programas reais. Nosso trabalho difere desses outros em propósito: estamos interessados em prevenir ataques de não terminação; e em método: nós instrumentamos somente uma pequena parte dos programas alvo.

Finalmente, nosso trabalho relaciona-se com outros que também procuram detectar, estaticamente, a não-terminação de programas. A maior parte desses trabalhos utilizam análise simbólica de código para criar expressões que levem um laço à não terminação. Exemplos desse tipo de pesquisa incluem os trabalhos de Burnim *et al.* [4], Brockschmidt *et al* [2] e Veroyen *et al.* [17]. Esses trabalhos não levam em consideração possibilidade de não-terminação devido à estouros de precisão, tampouco procuram detectar possíveis vulnerabilidades baseadas em negação de serviço. Existem, contudo, trabalhos na linha de detecção de não-terminação que são bastante próximos do nosso.

Um trabalho que prova não-terminação, mesmo em face de estouros de precisão deve-se à Gupta *et al.* [9]. Gupta, assim como os trabalhos anteriormente relacionados, utiliza análise simbólica para provar a não-terminação de programas. A ferramenta implementada por Gupta *et al.*, denominada TNT, é capaz de encontrar uma expressão algébrica que leva um laço de programa a iterar para sempre. Porém, TNT não aponta quais laços podem ser controlados a partir da entrada do programa. Por outro lado, a ferramenta SAFERPHP, proposta por Son *et al.* [15] possui exatamente esse objetivo. SAFERPHP analisa o código de programas escritos em PHP, procurando por laços que um adversário pode controlar, com o propósito, justamente, de evitar ataques de não-terminação. A principal diferença entre nosso trabalho, e aquele de Son *et al.*, é que, enquanto nossa ferramenta busca detectar a não-terminação devido à estouros de precisão inteira, SAFERPHP considera a aritmética de precisão infinita. Além disso, tanto SAFERPHP quanto TNT utilizam execução simbólica sobre caminhos possíveis no programa alvo. Essa abordagem, em nossa opinião, não é prática. Testemunho disso é o fato de tais ferramentas terem sido usadas, até a presente data, somente para analisar programas muito pequenos.

6 Conclusão

Referências

1. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
2. Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and nullpointerexceptions for java bytecode. In *FoVeOOS*, pages 123–141. Springer-Verlag, 2012.
3. David Brumley, Dawn Xiaodong Song, Tzi cker Chiueh, Rob Johnson, and Huijia Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS*. USENIX, 2007.
4. Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, pages 161–169. IEEE, 2009.
5. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.
6. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, 1977.
7. Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *ICSE*, pages 760–770. IEEE, 2012.
8. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.
9. Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. *SIGPLAN Not.*, 43(1):147–158, 2008.
10. C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *ISSSE*, pages 1–10. IEEE, 2006.
11. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.
12. Elias Levy. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
13. David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, 2006.
14. Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Efficient static checker for tainted variable attacks. In *SBLP*. SBC, 2010.
15. Sooel Son and Vitaly Shmatikov. SAFERPHP: finding semantic vulnerabilities in php applications. In *PLAS*, pages 8:1–8:13. ACM, 2011.
16. Omer Tripp, Marco Pistoia, Stephen Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *PLDI*, pages 87–97. ACM, 2009.
17. Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *TAP*, pages 154–170. Springer-Verlag, 2008.
18. Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41. ACM, 2007.