Instrumentação de programas binários legados para compatibilização com Intel CET

Éverton C. Araújo[1], **Mateus Tymburibá**[1], **Andrei Rimsa**[1]

[1]Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte – MG – Brasil

araujocostaeverton@gmail.com,{mateustymbu,andrei}@cefetmg.br

Resumo. *Mecanismos de proteção contra ataques que alteram o fluxo de execução de programas têm sido desenvolvidos por fabricantes de semicondutores. A tecnologia CET (Control-flow Enforcement Technology), criada recentemente pela Intel, é um exemplo promissor desse tipo de proteção. Para usá-la, programas devem incorporar uma nova instrução adicionada à arquitetura x86. Portanto, essa solução não se aplica a programas que não podem ser recompilados. Este trabalho propõe a utilização de técnicas de instrumentação para identificar as posições de um binário onde essa nova instrução deve ser adicionada. Posteriormente, ela é embutida nesses pontos identificados, utilizando uma técnica de remendo que garanta a integridade do executável.*

Abstract. *Protection mechanisms against attacks that alter the execution flow of programs have been developed by semiconductor manufacturers. The CET (Control-flow Enforcement Technology) technology, recently created by Intel, is a promising example of this type of protection. To use it, programs must incorporate a new instruction added to the x86 architecture. Therefore, this solution does not apply to programs that cannot be recompiled. This work proposes the use of instrumentation techniques to identify the positions in a binary where this new instruction must be added. Subsequently, it is embedded at these identified points, using a patching technique that ensures the integrity of the executable.*

1. Introdução

O crescimento de explorações de vulnerabilidades baseadas em injeção de código na memória tem impulsionado o desenvolvimento e a evolução de mecanismos de defesa contra invasões desse tipo. Embora recursos como ASLR (*Address Space Layout Randomization*) e DEP (*Data Execution Prevention*) ofereçam proteção contra diversos tipos de ataques, existem técnicas capazes de contornar essas defesas para permitir que invasores manipulem o fluxo de execução de um programa a seu favor. Três dessas técnicas, em particular, despertaram o interesse de fabricantes de semicondutores que estão colaborando na busca de soluções contra esses tipos de ataques. São elas: Programação Orientada a Retorno (ROP — *Return Oriented Programming*), Programação Orientada a Salto (JOP — *Jump Oriented Programming*) e Programação Orientada a Chamada (COP — *Call Oriented Programming*).

Em 2016, a Intel publicou a primeira versão das especificações de sua nova tecnologia, Intel CET (*Control-flow Enforcement Technology*), que combina os recursos de

pilha sombra (*shadow stack*) e rastreio de desvios indiretos (*indirect branch tracking*) em um único processador. O primeiro recurso tem a finalidade de mitigar ataques baseados em ROP, enquanto o segundo, ataques baseados em JOP e COP. Para a implementação desse último recurso, uma nova instrução foi introduzida no conjunto de instruções da arquitetura x86. Ela é utilizada para marcar os endereços de destino válidos para instruções de salto indireto (*jmp* e *call*). Essa instrução é chamada de ENDBRANCH e corresponde aos mnemônicos `endbr32` e `endbr64` para arquiteturas de 32 e 64 bits, respectivamente. Em 2020, a Intel anunciou que essa tecnologia estaria presente em seus processadores da linha *Tiger Lake*.

Softwares legados, nos quais não se tem mais acesso ao código-fonte para recompilação, mesmo em ambientes com suporte à tecnologia CET, permanecem vulneráveis a ataques. O recurso de rastreamento de saltos indiretos proposto pela tecnologia torna-se inviável, uma vez que a instrução ENDBRANCH não está presente no programa executável. Portanto, surge a necessidade de instrumentar esses binários para torná-los compatíveis com a tecnologia e eliminar essa fragilidade. É nesse contexto que este trabalho se insere. Seu objetivo principal é demonstrar a viabilidade da instrumentação de binários legados para torná-los compatíveis com a tecnologia Intel CET. Para isso, são realizadas análises estáticas e dinâmicas para identificar pontos no fluxo de controle dos programas que fariam uso dessa nova proteção. Para análise estática, utilizou-se o utilitário do Linux *objdump*. Para a análise dinâmica, usou-se a ferramenta *CFGgrind* [Rimsa et al. 2021]. Essas análises são complementares: a análise dinâmica identifica mais precisamente os pontos da execução onde ocorrem desvios indiretos, enquanto a análise estática identifica melhor os desvios em trechos não exercitados pela execução dinâmica. Uma vez identificados os possíveis endereços alvo desses desvios, altera-se o binário para incluir a instrução ENDBRANCH nesses pontos de forma a não modificar a semântica do programa. A viabilidade desses remendos é comprovada por meio de uma prova de conceito construída para um programa de exemplo, disponível em um repositório público do Github[1]. Essa demonstração tem como objetivo evidenciar que a instrumentação proposta é efetiva e aplicável em cenários reais.

2. Conceitos Preliminares

Nesta seção, serão mostradas as técnicas de ataque ROP, JOP e COP para explorar vulnerabilidas e como a tecnologia Intel CET atua para mitigar esses ataques.

2.1. ROP — *Return Oriented Programming*

ROP é uma técnica de desenvolvimento de explorações (*exploits*) que se baseia no encadeamento de pequenos trechos de código chamados de "*gadgets*", com o objetivo de assumir o controle do fluxo de uma aplicação e executar ações desejadas pelo atacante [Ferreira et al. 2012]. Essa técnica é utilizada para contornar uma forma de proteção de memória conhecida como NX/XD (*No eXecute* em processadores da AMD e *eXecute Disable* em processadores da Intel), que impede a execução de instruções em áreas marcadas com o bit de proteção.

Os *gadgets* em um ataque ROP consistem em sequências curtas de instruções coletadas a partir da área de memória executável, como o código fonte compilado e módulos

[1] `https://github.com/araujoec/instr-to-cet-compat`

compartilhados, finalizadas com uma instrução de retorno (*ret*). O uso de *gadgets* tem como objetivo fazer com que a própria aplicação execute as operações necessárias para o *exploit*, por meio de reuso de código.

A Figura 1 ilustra um exemplo de um ataque ROP em execução, onde um valor de $0x32400$ é adicionado ao valor armazenado no endereço *0x4a304120*. Esse *exploit* começa carregando o valor do endereço *0x4a304120* no registrador EAX e o valor $0x32400$ no registrador EBX. Em seguida, o valor contido no endereço fornecido é armazenado em EAX. Os valores de EAX e EBX são somados e armazenados em EAX. O endereço inicial é armazenado em ECX e, por fim, o resultado da soma em EAX é copiado para o endereço armazenado em ECX. A escolha e o encadeamento desses *gadgets* é uma parte crucial e cautelosa dessa técnica para o sucesso do ataque, uma vez que a execução de qualquer instrução adicional ou fora de ordem pode levar a resultados inesperados no *exploit*.

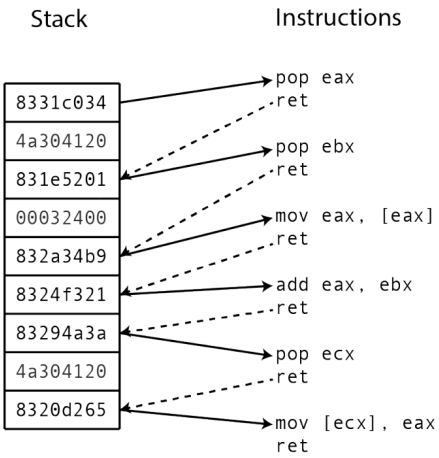


Figura 1. Exemplo de um *exploit* ROP [Carlini and Wagner 2014].

2.2. JOP — *Jump Oriented Programming*

A técnica JOP é semelhante ao ROP no sentido de que também envolve o encadeamento de *gadgets*. No entanto, os *gadgets* utilizados no JOP consistem em conjuntos de instruções finalizadas com instruções de salto indireto (*jmp*). Uma diferença importante em relação ao ROP é que, enquanto em um ataque ROP um *gadget* retorna naturalmente o controle para a aplicação com base no conteúdo da pilha, em um ataque JOP a transferência de controle é unidirecional para o endereço de destino do próximo *gadget*. Isso torna mais difícil a recuperação do controle do fluxo e o encadeamento de *gadgets* no JOP.

Existem algumas variações de ataques do tipo JOP, sendo a mais popular baseada no conceito de "*dispatcher gadget*" (*gadget* despachante) [Bletsch et al. 2011]. Esse *gadget* despachante é responsável por gerenciar o fluxo de controle entre os demais *gadgets*, conhecidos como "*functional gadgets*" (*gadgets* funcionais), determinando qual *gadget* funcional será invocado em sequência. Para viabilizar esse mecanismo, é criada uma "*dispatcher table*" (tabela despachante), que lista os endereços dos *gadgets* funcionais, bem como os dados necessários para a execução das operações desejadas. Além disso, um registrador qualquer, não necessariamente o que aponta para o topo da pilha (ESP),

é utilizado para apontar para essa tabela, e esse registrador é mantido como um "*virtual program counter*" (contador de programa virtual).

Em cada etapa do ataque JOP, o *gadget* despachante incrementa o contador de programa virtual e executa o *gadget* associado ao endereço indicado na tabela. Esse processo de encadeamento de *gadgets* por meio da tabela despachante permite a realização de um ataque de reuso de código que não depende explicitamente de instruções de retorno (*ret*), possibilitando, assim, superar medidas de proteção que monitorem a pilha ou instruções de retorno, como é o caso da técnica da pilha sombra.

No exemplo ilustrado na Figura 2, os registradores ESI e EDI são usados para retornar o controle de fluxo para o *dispatcher gadget*. A sequência de saltos é indicada pelos números de 1 a 6 na figura e a descrição a seguir detalha cada etapa:

1. o primeiro salto invoca um *gadget* responsável por carregar um valor da memória para um registrador;
2. após carregar o valor do endereço armazenado em EAX para o próprio registrador, ocorre o retorno ao *dispatcher gadget*;
3. o terceiro salto invoca um *gadget* de adição de dois registradores;
4. nessa etapa, o registrador EBX é "derreferenciado" e o seu valor é adicionado a EAX. Em seguida, ocorre o retorno ao *dispatcher gadget*;
5. o quinto salto invoca um *gadget* de armazenamento em memória;
6. nessa última etapa, o valor contido em EAX é salvo na memória, e novamente ocorre o retorno ao *dispatcher gadget*.

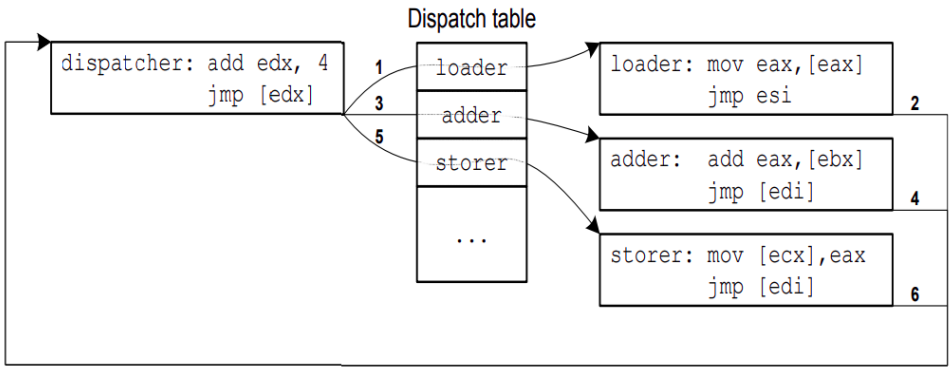


Figura 2. Controle de fluxo em um exemplo de *exploit* JOP [Bletsch et al. 2011].

2.3. COP — *Call Oriented Programming*

A técnica COP segue uma lógica similar às técnicas ROP e JOP, utilizando *gadgets* para o reuso de código. No entanto, ao contrário das técnicas anteriores, os *gadgets* utilizados na técnica COP são finalizados com instruções de chamada de procedimento (*call*). A técnica COP pode parecer semelhante à JOP por também se basear em instruções de salto indireto [Carlini and Wagner 2014]. Porém, há uma distinção importante entre elas: "chamadas de função indiretas usualmente acessam a memória, ao invés de registradores". Isso significa que o controle de fluxo não é transferido para um valor específico em um registrador, mas sim para um valor armazenado em uma localização de memória. Portanto, não é necessário o uso de um *dispatcher gadget* como no ataque JOP. Para encadear os *gadgets* na técnica COP, eles devem apontar para locais de memória que contêm o

endereço do próximo *gadget* na sequência, sendo possível realizar a inicialização desses locais de memória antecipadamente.

2.4. CET — *Control-Flow Enforcement Technology*

As técnicas descritas nas seções anteriores são usadas para corromper a integridade do fluxo de execução de um programa por meio de seu sequestro. Diversos trabalhos foram propostos com o intuito de evitar esse tipo de subversão do fluxo de execução de programas [Moreira et al. 2017, Tymburibá et al. 2019, Botacin et al. 2016, Ferreira et al. 2014, Emílio et al. 2015]. Entre os mecanismos de proteção já propostos, a tecnologia Intel CET tem se mostrado altamente promissora no combate a ameaças comuns de malware [Intel 2019]. Trata-se do estado da arte em mecanismos de proteção contra ataques de reuso de código. Anunciada em junho de 2020 pela Intel, essa tecnologia começou a ser disponibilizada em processadores móveis da linha *Tiger Lake*. Ela faz uso de dois recursos principais para implementação da tecnologia: o rastreamento de saltos indiretos (IBT – *indirect branch tracking*) e a pilha sombra (SS - *shadow stack*).

O recurso de pilha sombra consiste em uma segunda pilha separada da área de pilha do processo, que armazena os endereços de retorno das funções. Essa pilha é dedicada exclusivamente a operações de controle de fluxo e é protegida contra adulterações por meio de uma extensão da tabela de páginas que marca essa área como "*shadow stack*". Dessa forma, operações de armazenamento comuns (*store*) não podem modificar o conteúdo dessa pilha. A escrita na pilha sombra é permitida apenas indiretamente por meio de instruções de transferência de controle e manipulações específicas da pilha sombra, as quais só podem ser executadas por código com nível de privilégio elevado. As instruções de retorno removem elementos tanto da pilha sombra quanto da pilha regular e comparam seus valores. Se os endereços comparados não forem iguais, o processador lança uma exceção de proteção de controle (CP — *Control Protection Exception*) [Intel 2019].

Para o recurso de rastreamento de saltos indiretos, o processador implementa uma máquina de estados que monitora as instruções indiretas de salto (*jmp*) e chamada (*call*), conforme ilustrado na Figura 3 [Shanbhogue et al. 2019]. Quando uma dessas instruções é executada, o estado da máquina muda de "IDLE" para "WAIT_FOR_ENDBRANCH". Nesse estado, a próxima instrução a ser executada deve ser uma instrução ENDBRANCH. Caso contrário, o processador lança a exceção de proteção de controle. Se a instrução ENDBRANCH for encontrada, a máquina de estado retorna ao estado "IDLE". A instrução ENDBRANCH é a nova instrução introduzida para marcar os destinos de saltos indiretos e chamadas de funções indiretas. Elas são representadas em processadores de arquiteturas de 32 bits e de 64 bits, respectivamente, pelas instruções "*endbr32*" e "*endbr64*". Entretanto, em processadores mais antigos, que não possuem suporte a essa tecnologia, ENDBRANCH corresponde a uma instrução "*nop*" (*no operation*). Isso faz com que essa tecnologia seja retrocompatível.

3. Ferramentas utilizadas

O processo de instrumentação dos programas binários envolveu o uso das ferramentas descritas nas subseções 3.1 e 3.2.

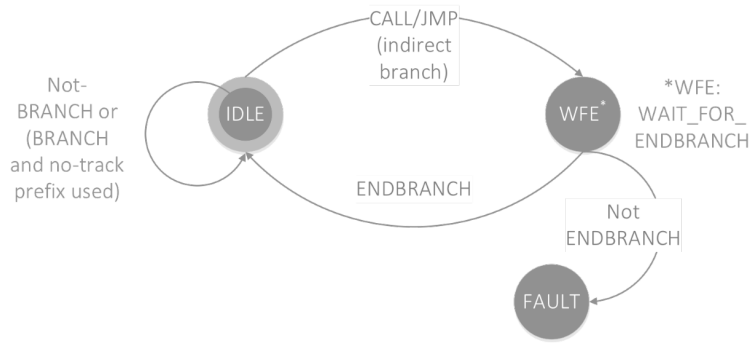


Figura 3. Máquina de estado para recurso IBT [Shanbhogue et al. 2019].

3.1. *objdump*

objdump é uma ferramenta de linha de comando em sistemas operacionais *Unix* para exibir informações sobre um arquivo objeto (*object file*). Basicamente, um *object file* em formato ELF é composto por um cabeçalho ELF (*ELF header*), seguido por uma tabela de cabeçalho de programa (*program header table*), ou uma tabela de cabeçalho de seções (*section header table*), ou ambas. O *ELF header* está sempre no endereço (*offset*) zero do arquivo e define a estrutura das tabelas.

Neste trabalho, é utilizada a opção $-d$ desse comando, que permite exibir todas as instruções em código de máquina de um determinado programa. A partir delas é possível identificar a existência e localização de instruções de desvio.

3.2. *CFGgrind*

A ferramenta *CFGgrind* [Rimsa et al. 2021] permite a reconstrução dinâmica do grafo de fluxo de controle (CFG) a partir da sequência de execução de instruções de um binário. Ela é implementada como um plugin do *Valgrind* e permite sucessivos refinamentos de CFGs, pois suporta múltiplas execuções com diferentes entradas. Portanto, para obter um CFG completo, é necessário fornecer entradas que exercitem todas as instruções possíveis para o programa.

Com base nas informações extraídas durante a execução do programa e embutidas no CFG, é possível identificar os endereços de destino das instruções de desvio indireto que são tomadas durante a execução do programa. Isso torna possível identicar esses pontos críticos para determinar o local exato onde deve ocorrer a inserção da instrução ENDBRANCH. Vale ressaltar que o problema de identificar todos os pontos de desvios indiretos para programas arbitrários é indecidível, segundo o teorema de Rice.

4. Estratégia de instrumentação

A estratégia de instrumentação adotada neste trabalho baseia-se no princípio fundamental da ferramenta de reescrita estática de binários *E9Patch* [Duck et al. 2020]. Essa ferramenta utiliza um conjunto de metodologias de reescrita (*instruction punning*, *padding* e *eviction*) para inserir saltos (*jumps*) para seus "trampolins" sem a necessidade de mover as demais instruções, tornando a solução "inofensiva" ao fluxo de controle original.

Neste trabalho, o artifício do trampolim é adotado de uma maneira simplificada, sem a utilização de todas as táticas de reescrita. A instrumentação consiste em duas partes,

denominadas aqui de *remendo* e *trampolim*, com o objetivo de facilitar a diferenciação entre elas. O remendo é um trecho de código que sobrescreve as instruções no endereço desejado com uma instrução ENDBRANCH seguida de uma instrução *jump* para um trampolim e, quando necessário, instruções *nop* para preenchimento do restante da instrução sobrescrita. O trampolim, por sua vez, é formado pelas instruções que o remendo sobrescreve, seguidas de uma instrução *jmp* que retorna para a instrução imediatamente após o remendo. A Figura 4 ilustra de forma esquemática o remendo e o trampolim.

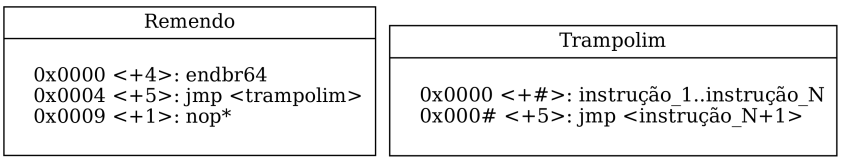


Figura 4. Estratégia de instrumentação.

A parte fundamental do remendo corresponde às instruções *endbr64* e *jmp*, com tamanhos de 4 e 5 bytes, respectivamente, totalizando 9 bytes. O asterisco ("*") na instrução *nop* indica que essa instrução pode aparecer zero ou mais vezes. Isso ocorre porque, caso a parte fundamental do remendo não coincida com o fim de uma instrução sobrescrita, os bytes de não-operação completam o espaço originalmente ocupado pela instrução sobrescrita. No trampolim, o trecho "instrução_1..instrução_N" representa todas as instruções sobrescritas pelo remendo, porém adaptadas a seus novos endereços. Já a "instrução_N+1" representa a instrução logo após o remendo. Não será possível efetuar o remendo caso não haja espaço suficiente para ele dada a limitação de espaço de 9 bytes.

Para construir os trampolins, sugere-se o uso de 'áreas livres' do binário. Essas áreas são espaços do arquivo que não contêm segmentos de instruções ou informações relacionadas ao programa. Por exemplo, em arquivos no formato ELF, a área entre as seções *.fini* e *.rodata* costuma ter uma grande quantidade de bytes disponíveis para os trampolins.

Embora nos experimentos realizados neste trabalho não se tenha verificado desvios para instruções sobrescritas pelo remendo, é importante mencionar a possibilidade de ocorrência dessa situação em cenários específicos. Para tratar situações como essa, as táticas de reescrita apresentadas pela ferramenta *E9Patch* podem ser utilizadas.

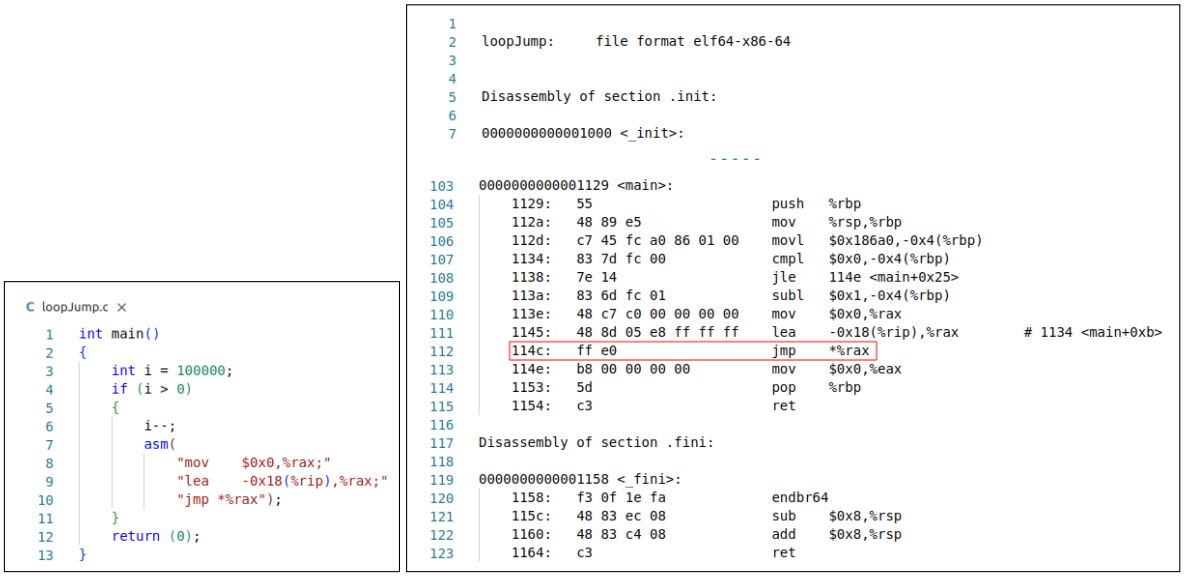
5. Prova de Conceito (*PoC*)

Um exemplo de programa, escrito em linguagem C, foi implementado para ilustrar o funcionamento da estratégia proposta. O código é compilado pelo compilador GCC por meio da linha de comandos, com a opção de desabilitar as proteções de fluxo de controle, "*-fcf-protection=none*", que o próprio compilador oferece. Como algumas funções são de bibliotecas compartilhadas previamente compiladas, elas podem conter a instrução ENDBRANCH, mesmo com a opção mencionada desativada.

Embora não sejam casos reais de binários legados, os experimentos realizados neste trabalho ainda se aplicam a esses binários, pois a instrumentação pode ser executada em qualquer programa compilado na arquitetura *x86*.

5.1. Desvio indireto com *jump*

O código foi implementado com uso de código *assembly inline* para forçar a existência de uma instrução d *jmp* indireto no meio. Basicamente, um *loop* com 100 mil iterações é executado até zerar o valor da variável *i*. As Figuras 5 (a) e (b) mostram o código-fonte e a saída do comando *objdump* para seu arquivo objeto compilado.



loopJump.c

```c
int main()
{
    int i = 100000;
    if (i > 0)
    {
        i--;
        asm(
            "mov    $0x0,%rax;"
            "lea    -0x18(%rip),%rax;"
            "jmp *%rax");
    }
    return (0);
}
```

```
loopJump:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
                                -----
0000000000001129 <main>:
    1129:   55                      push   %rbp
    112a:   48 89 e5                mov    %rsp,%rbp
    112d:   c7 45 fc a0 86 01 00    movl   $0x186a0,-0x4(%rbp)
    1134:   83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
    1138:   7e 14                   jle    114e <main+0x25>
    113a:   83 6d fc 01             subl   $0x1,-0x4(%rbp)
    113e:   48 c7 c0 00 00 00 00    mov    $0x0,%rax
    1145:   48 8d 05 e8 ff ff ff    lea    -0x18(%rip),%rax        # 1134 <main+0xb>
    114c:   ff e0                   jmp    *%rax
    114e:   b8 00 00 00 00          mov    $0x0,%eax
    1153:   5d                      pop    %rbp
    1154:   c3                      ret

Disassembly of section .fini:

0000000000001158 <_fini>:
    1158:   f3 0f 1e fa             endbr64
    115c:   48 83 ec 08             sub    $0x8,%rsp
    1160:   48 83 c4 08             add    $0x8,%rsp
    1164:   c3                      ret
```

(a) Programa *loopJump* em C. (b) Saída simplificada do comando "*objdump -dh loopJump*".

Figura 5. Código fonte de *loopJump* e representação de seu arquivo executável.

A instrução de salto indireto definida pelo código-fonte está no endereço *114c*, conforme destacado na Figura 5 (b). Neste exemplo específico, é possível identificar o endereço de destino da instrução "*jmp *%rax*" pela análise estática com o *objdump*, a partir do cálculo realizado pela instrução anterior. No entanto, a análise dinâmica será demonstrada, para fins didáticos. Quando o programa é analisado pelo *CFGgrind* e seu grafo de fluxo de controle é gerado, o endereço de destino encontrado é também *1134*, como mostra a Figura 6.

Ao contar os 9 primeiros bytes a partir desse endereço, tem-se as instruções "*cmpl $0x0,-0x4(%rbp)*", "*jle 114e*" e "*subl $0x1, -0x4(%rbp)*" (sendo a última instrução considerada até o final, conforme mencionado na seção 4). Essas instruções serão sobrescritas pelo remendo e reescritas no trampolim. Como observado também na Figura 5 (b), a última instrução da seção *.fini* finaliza no endereço *1164*. Isso significa que o trampolim desse remendo pode ser escrito a partir do endereço *1165*. Dessa forma, todas as informações necessárias para fazer a instrumentação desse programa estão disponíveis, conforme representado esquematicamente na Figura 7. Nessa figura, as linhas em vermelho, iniciadas pelo sinal "-", representam as instruções que serão substituídas, na mesma ordem, pelas linhas em verde, iniciadas com o sinal "+". Um detalhe importante é o ajuste nos bytes da instrução "*jle 114e*", a qual, no endereço *1138*, era "*7e 14*" e passou a ser "*7e e3*", para manter o destino do salto no endereço *114e*.

Para modificar o código binário, são criados dois arquivos com os bytes das

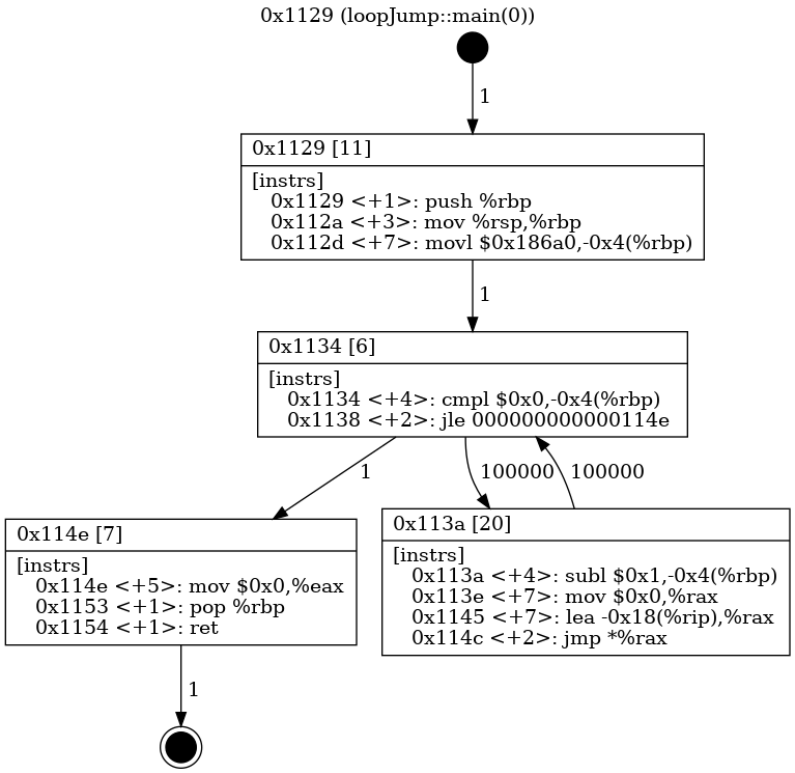


Figura 6. Grafo de Fluxo de Controle da função *main* do programa *loopJump*.

```
106  106       112d:   c7 45 fc a0 86 01 00   movl   $0x186a0,-0x4(%rbp)
107       -    1134:   83 7d fc 00            cmpl   $0x0,-0x4(%rbp)
108       -    1138:   7e 14                  jle    114e <main+0x25>
109       -    113a:   83 6d fc 01            subl   $0x1,-0x4(%rbp)
     107  +    1134:   f3 0f 1e fa            endbr64
     108  +    1138:   e9 28 00 00 00         jmp    1165 <_fini+0xd>
     109  +    113d:   90                     nop
110  110       113e:   48 c7 c0 00 00 00 00   mov    $0x0,%rax
```

(a) Remendo

```
121  121       115c:   48 83 ec 08            sub    $0x8,%rsp
122  122       1160:   48 83 c4 08            add    $0x8,%rsp
123  123       1164:   c3                     ret
     124  +
     125  +    1165:   83 7d fc 00            cmpl   $0x0,-0x4(%rbp)
     126  +    1169:   7e e3                  jle    114e <main+0x25>
     127  +    116b:   83 6d fc 01            subl   $0x1,-0x4(%rbp)
     128  +    116f:   e9 ca ff ff ff         jmp    113e <main+0x15>
```

(b) Trampolim

Figura 7. Representação esquemática da instrumentação para o programa *loopJump*.

instruções correspondentes a cada modificação, representados em hexadecimal. Os comandos a seguir criam os arquivos *remendo.bin* e *trampolim.bin*:

```
$ echo -e -n     "\xf3\x0f\x1e\xfa\xe9\x28\x00\x00\x00\x90" \
      > remendo.bin
$ echo -e -n "\x83\x7d\xfc\x00\x7e\xe3\x83\x6d\xfc\x01" \
  "\xe9\xca\xff\xff\xff"  > trampolim.bin
```

Com esses arquivos prontos, é possível usar a ferramenta *dd* para sobrescrever o arquivo binário original com os valores que foram preparados para a instrumentação. O arquivo do remendo deve começar a sobrescrita a partir do endereço *1134*, que na base decimal equivale a *4404*, enquanto o arquivo do trampolim deve iniciar no endereço *1165*, ou *4453* em decimal. Os comandos a seguir fazem a modificação no arquivo do programa:

```
$ dd if=remendo.bin of=loopJump bs=1 count=10 seek=4404 conv=notrunc
$ dd if=trampolim.bin of=loopJump bs=1 count=15 seek=4453 conv=notrunc
```

O remendo pode ser observado pelo *objdump*, pois encontra-se dentro de uma seção que é mapeada pela ferramenta. No entanto, o trampolim não é exibido dessa forma. Para visualizá-lo, usou-se a ferramenta *xxd*. A Figura 8 compara a diferença do binário antes e depois da instrumentação por meio das saídas do *xxd*. Na linha 276 observa-se o remendo (entre os endereços *1134* e *113d*), e na linha 279 vê-se o início do trampolim (entre os endereços *1165* e *1173*).

```
273  273    00001100: 0000 e829 ffff ffe8 64ff ffff c605 fd2e  ...)....d.......
274  274    00001110: 0000 015d c30f 1f00 c30f 1f80 0000 0000  ...]............
275  275    00001120: f30f 1efa e977 ffff ff55 4889 e5c7 45fc  .....w...UH...E.
276       - 00001130: a086 0100 837d fc00 7e14 836d fc01 48c7  .....}..~..m..H.
     276  + 00001130: a086 0100 f30f 1efa e928 0000 0090 48c7  .........(....H.
277  277    00001140: c000 0000 0048 8d05 e8ff ffff ffe0 b800  .....H..........
278  278    00001150: 0000 005d c300 0000 f30f 1efa 4883 ec08  ...]........H...
279       - 00001160: 4883 c408 c300 0000 0000 0000 0000 0000  H...............
280       - 00001170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
     279  + 00001160: 4883 c408 c383 7dfc 007e e383 6dfc 01e9  H.....}..~..m...
     280  + 00001170: caff ffff 0000 0000 0000 0000 0000 0000  ................
281  281    00001180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
282  282    00001190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
283  283    000011a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Figura 8. Bytes modificados após instrumentação.

Para confirmar que o programa faz o salto para o trampolim e retorna ao fluxo original, utilizou-se o depurador *gdb*. É necessário definir um *breakpoint* no início da função *recursao* e executar cada instrução passo a passo. A Figura 9 mostra os comandos e as instruções executadas em cada passo, sendo que as instruções relacionadas ao remendo e ao trampolim estão sublinhadas em vermelho. O desvio indireto da instrução *jmp* é executado e encontra logo em seguida a instrução *endbr64* no endereço alvo. Os endereços em azul, à esquerda de cada instrução, mostram a sequência do ponteiro de instrução passando pelo remendo, em seguida pelo trampolim, e retornando para a instrução logo após o remendo.

6. Considerações finais

Este trabalho demonstra a viabilidade de instrumentação de um binário legado para compatibilização com a tecnologia Intel CET. A estratégia sugerida foi implementada por meio de remendos e trampolins que sobrescrevem bytes em endereços específicos e desviam o fluxo de controle do programa, retornando ao fluxo original logo em seguida.

Quando comparada ao procedimento de mover todas as instruções subsequentes a um ponto de inserção para a adição de novos códigos, a estratégia de instrumentação apresentada neste trabalho oferece vantagens relacionadas a uma menor complexidade de implementação e à manutenção do tamanho do arquivo binário final. Uma das facilidades

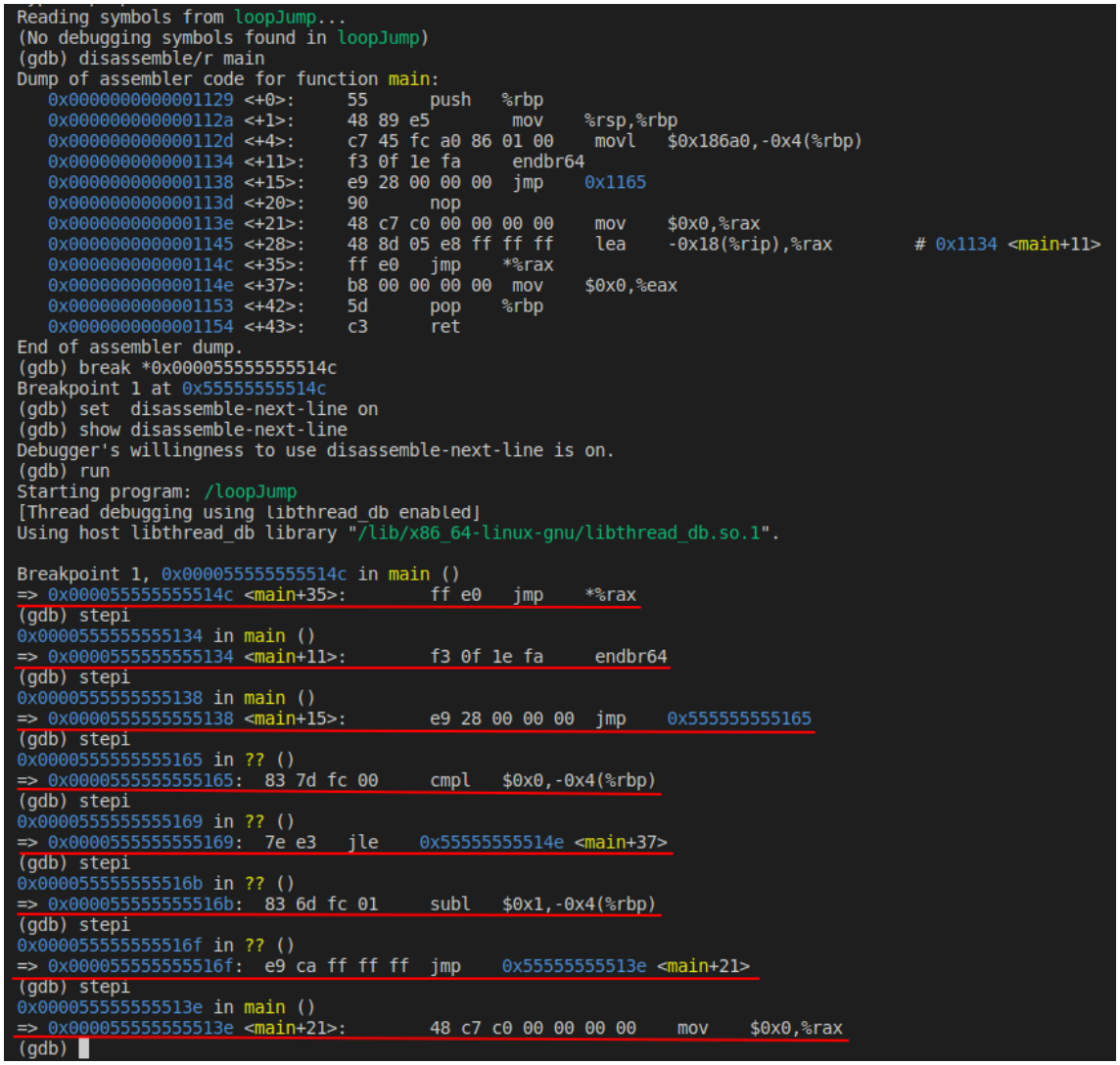


Figura 9. Instruções executadas em *loopJump* após instrumentação, passando pelo remendo e pelo trampolim.

relacionadas à complexidade é o uso de ferramentas simples e nativas de sistemas operacionais *Unix* para implementar a modificação do binário. Apenas a ferramenta de análise dinâmica de binário, o *CFGgrind*, não é nativa. Além disso, a estragégia se baseia em construções simples de trampolins em áreas livres do próprio binário. Isso se reflete na outra vantagem: o tamanho do arquivo instrumentado permanece o mesmo do arquivo original, já que há apenas sobrescrição de bytes, sem acréscimos.

Em situações específicas, contudo, essa estratégia pode demandar ajustes. Suponha, por exemplo, dois endereços de destino distantes a menos de 9 bytes um do outro, como pode ocorrer em estruturas do tipo *switch case*. Quando os remendos para esses endereços forem construídos e sobrescritos, inevitavelmente ocorrerá um conflito, uma vez que são necessários no mínimo 9 bytes para cada remendo. Por fim, considerando que os procedimentos de instrumentação adotados são bem definidos, em trabalhos futuros pretende-se automatizar esses processos de inserção de remendos e trampolins.

7. Agradecimento

Agradecemos ao CEFET-MG pelo apoio durante o desenvolvimento deste trabalho.

Referências

[Bletsch et al. 2011] Bletsch, T. K., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. In Cheung, B. S. N., Hui, L. C. K., Sandhu, R. S., and Wong, D. S., editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 30–40. ACM.

[Botacin et al. 2016] Botacin, M., de Geus, P. L., and Grégio, A. (2016). Detecção de ataques por rop em tempo real assistida por hardware. In *Anais do XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 324–337. SBC.

[Carlini and Wagner 2014] Carlini, N. and Wagner, D. (2014). {ROP} is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, pages 385–399.

[Duck et al. 2020] Duck, G. J., Gao, X., and Roychoudhury, A. (2020). Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA. Association for Computing Machinery.

[Emílio et al. 2015] Emílio, R., Tymburibá, M., and Pereira, F. M. Q. (2015). Inferência estática da frequência máxima de instruções de retorno para detecção de ataques rop. In *Anais do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 2–15. SBC.

[Ferreira et al. 2012] Ferreira, M., Rocha, T., Martins, G., Feitosa, E., and Souto, E. (2012). Análise de vulnerabilidades em sistemas computacionais modernos: Conceitos, exploits e proteções.

[Ferreira et al. 2014] Ferreira, M. T., Santos Filho, A., and Feitosa, E. (2014). Controlando a frequência de desvios indiretos para bloquear ataques rop. In *Anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 223–236. SBC.

[Intel 2019] Intel (2019). Control-flow enforcement technology specification. `https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf`.

[Moreira et al. 2017] Moreira, J., Rigo, S., Polychronakis, M., and Kemerlis, V. P. (2017). Drop the rop fine-grained control-flow integrity for the linux kernel. *Black Hat Asia*, 2017.

[Rimsa et al. 2021] Rimsa, A., Nelson Amaral, J., and Pereira, F. M. Q. (2021). Practical dynamic reconstruction of control flow graphs. *Software: Practice and Experience*, 51(2):353–384.

[Shanbhogue et al. 2019] Shanbhogue, V., Gupta, D., and Sahita, R. (2019). Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA. ACM.

[Tymburibá et al. 2019] Tymburibá, M., de Sousa, H. A., and Pereira, F. M. Q. (2019). Multilayer rop protection via microarchitectural units available in commodity hardware. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 315–327. IEEE.