

UNIVERSIDADE FEDERAL DO AMAZONAS

Programa de Pós-Graduação em Informática

BRUNO ROCHA RIBEIRO

**COMPATIBILIZAÇÃO DE BINÁRIOS LEGADOS
COM A PROTEÇÃO DE RASTREAMENTO DE
DESVIOS INDIRETOS**

MANAUS
2026

BRUNO ROCHA RIBEIRO

**COMPATIBILIZAÇÃO DE BINÁRIOS LEGADOS
COM A PROTEÇÃO DE RASTREAMENTO DE
DESVIOS INDIRETOS**

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas como requisito parcial para obtenção do grau de Mestre em Informática, área de concentração: Cibersegurança.

ORIENTADOR: EDUARDO JAMES PEREIRA SOUTO
COORIENTADOR: MATEUS FELIPE TYMBURIBÁ FERREIRA

MANAUS
2026

Ficha Catalográfica

Elaborada automaticamente de acordo com os dados fornecidos pel(a) autor(a).

R484c Ribeiro, Bruno Rocha
Compatibilização de binários legados com a proteção de rastreamento de desvios indiretos / Bruno Rocha Ribeiro. - 2026.
125 f. : il., color. ; 31 cm.

Orientador(a): Eduardo James Pereira Souto.
Coorientador(a): Mateus Felipe Tymburibá Ferreira
Dissertação (mestrado) - Universidade Federal do Amazonas, Programa de Pós-Graduação em Informática, Manaus, 2026.

1. Reescrita binária estática. 2. Intel CET - Control-flow Enforcement Technology. 3. IBT - Indirect Branch Tracking. 4. Binários legados. 5. Integridade de fluxo de controle. I. Souto, Eduardo James Pereira. II. Ferreira, Mateus Felipe Tymburibá. III. Universidade Federal do Amazonas. Programa de Pós-Graduação em Informática IV. Título



Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"COMPATIBILIZAÇÃO DE BINÁRIOS LEGADOS COM A PROTEÇÃO DE RASTREAMENTO DE DESVIOS INDIRETOS"

BRUNO ROCHA RIBEIRO

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Dr. Eduardo James Pereira Souto - **Presidente**

Documento assinado digitalmente
gov.br JOAO BATISTA CORREA GOMES MOREIRA
Data: 07/09/2026 16:40:16-0300
Verifique em <https://validar.it.gov.br>

Dr. João Batista Correa Gomes Moreira - **Membro Externo**

Prof. Dr. Gilbert Breves Martins - **Membro Externo**

Manaus, 28 de Julho de 2026.



Documento assinado eletronicamente por **Gilbert Breves Martins, Usuário Externo**, em 29/07/2026, às 17:37, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Eduardo James Pereira Souto, Professor do Magistério Superior**, em 03/08/2026, às 09:39, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3255120** e o código CRC **27C46DBD**.

Avenida General Rodrigo Octávio, 6200 - Bairro Coroado I Campus Universitário Senador Arthur Virgílio Filho, Setor Norte - Telefone: (92) 3305-1181 / Ramal 1193
CEP 69080-900, Manaus/AM, coordenadorppgi@icomp.ufam.edu.br

Agradecimentos

Primeiramente, sou grato a Deus pelas oportunidades colocadas em minha vida. Desde o princípio fui abençoado com uma família amável e a cada dia sinto a presença do Senhor em minha vida, especialmente marcada por diversas oportunidades. A concretização desta pesquisa é uma delas, representando não apenas um marco acadêmico, mas também uma vitória construída ao longo de muitos anos. Agradeço profundamente à minha esposa, cuja presença constante ofereceu equilíbrio, tranquilidade e motivação diante dos desafios intelectuais e emocionais que emergiram durante o desenvolvimento deste trabalho. Sua compreensão nos momentos de ausência, sua paciência nos períodos de maior intensidade e seu incentivo diário foram fundamentais para que esta jornada se mantivesse sólida e coerente.

Manifesto também minha gratidão aos meus pais, que sempre sustentaram minha formação com valores que ultrapassam o âmbito acadêmico. Foram eles que fomentaram o hábito do estudo disciplinado, o respeito pela educação e a busca incessante por conhecimento. Sua confiança irrestrita e seu apoio contínuo criaram as condições necessárias para que eu pudesse trilhar um caminho inteiramente dedicado ao aprimoramento intelectual e profissional.

Estendo meus agradecimentos aos professores e pesquisadores que contribuíram para minha formação nas instituições que marcaram minha trajetória. No Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG), encontrei a base sólida que estruturou meu pensamento técnico e crítico, além do incentivo para explorar temas complexos com rigor metodológico. Na Universidade Federal do Amazonas (UFAM), tive a oportunidade de amadurecer academicamente, aprofundando meus interesses e sendo desafiado a desenvolver soluções originais para problemas contemporâneos. Cada orientação, discussão e crítica construtiva foi indispensável para o amadurecimento deste trabalho.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para o avanço desta pesquisa. Profissionais da área, colaboradores e amigos que ofereceram apoio moral, revisões cuidadosas e discussões que desempenharam papel essencial na consolidação deste resultado. A todos que caminharam ao meu lado, deixo registrado meu sincero reconhecimento.

O presente trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (AUXPE-CAPES-PROEX) - Código de Financiamento 001. Adicionalmente, este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas - FAPEAM - por meio dos projetos PDPG-CAPES e POSGRAD 2025-2026 e POSGRAD 2026-2027.

Resumo

A crescente sofisticação dos ataques por reutilização de código, como *Return-Oriented Programming* (ROP) e *Jump-Oriented Programming* (JOP), tem motivado a adoção de mecanismos mais robustos de proteção da integridade do fluxo de controle. Entre essas proteções, destaca-se a *Control-flow Enforcement Technology* (CET) da Intel, composta principalmente pelos mecanismos *Indirect Branch Tracking* (IBT) e *Shadow Stack*. Embora compiladores modernos já sejam capazes de gerar binários compatíveis com CET, uma parcela significativa de softwares legados permanece desprotegida por não poder ser recompilada, seja pela indisponibilidade do código-fonte, por restrições de manutenção ou por dependência de ambientes antigos. Este trabalho propõe o BIN2CET, um sistema automatizado de reescrita binária estática para compatibilizar binários ELF legados com o mecanismo IBT da Intel CET, sem exigir acesso ao código-fonte nem recuperação completa do grafo de fluxo de controle. A abordagem combina três estratégias complementares: inserção de instruções *endbr64* em alvos recuperados de desvios indiretos, preservação de instruções sobrescritas por meio de trampolins cientes de realocação e uso conservador de *notrack* quando a reescrita segura do alvo não é possível. O sistema utiliza a biblioteca LIEF para manipulação em nível de ELF e uma versão estendida do E9Patch para materialização de trampolins, redirecionamento de fluxo e preservação semântica de instruções sensíveis à posição, como referências relativas a RIP. A avaliação experimental foi conduzida sobre os 102 programas da suíte Coreutils, compilados como binários PIE (*Position-Independent Executable*) sob os níveis de otimização -O0, -O2 e -O3. Os resultados indicam cobertura completa dos alvos recuperados de entrada de função no cenário não otimizado e cobertura superior a 80% nos layouts otimizados. As medições de desempenho apontam *overhead* de execução entre aproximadamente 22% e 42% para os principais contadores de tempo do Linux *perf*, enquanto a análise de tamanho binário mostra crescimento estrutural compatível com a inserção de marcadores, redirecionamentos e trampolins. Além disso, a análise da superfície de ataque, conduzida com ROPgadget, indica que a superfície efetiva de transferências indiretas válida pela política é reduzida para menos de 1% do espaço sintático de *gadgets* identificados. Esses resultados indicam que a reescrita binária estática automatizada constitui um caminho viável para compatibilizar softwares legados com os requisitos do IBT, preservando funcionalidade dos executáveis e oferecendo uma base técnica para extensões futuras de endurecimento binário.

Palavras-chave: reescrita binária estática, Intel CET, IBT, binários legados, integridade de fluxo de controle, *endbr64*, *notrack*.

Abstract

The increasing sophistication of code-reuse attacks, such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP), has motivated the adoption of stronger mechanisms for protecting control-flow integrity. Among these protections, Intel’s Control-flow Enforcement Technology (CET) stands out, consisting primarily of Indirect Branch Tracking (IBT) and Shadow Stack. Although modern compilers can generate CET-compatible binaries, much legacy software remains unprotected because it cannot be recompiled, either due to unavailable source code, maintenance constraints, or dependence on older environments. This work proposes BIN2CET, an automated static binary rewriting system that makes legacy ELF binaries compatible with Intel CET’s IBT mechanism without requiring source code or complete control-flow graph recovery. The approach combines three complementary strategies: inserting `endbr64` instructions at recovered indirect-branch targets, preserving overwritten instructions through relocation-aware trampolines, and conservatively using `notrack` when safe target rewriting is not possible. The system uses LIEF for ELF-level manipulation and an extended version of E9Patch to materialize trampolines, redirect control flow, and preserve the semantics of position-sensitive instructions, such as RIP-relative references. The experimental evaluation was conducted on the 102 programs of the Coreutils suite, compiled as PIE (Position-Independent Executable) binaries under the `-O0`, `-O2`, and `-O3` optimization levels. The results indicate complete coverage of recovered function-entry targets in the unoptimized scenario and coverage above 80% in optimized layouts. Performance measurements show execution overhead between approximately 22% and 42% for the main Linux `perf` timing counters, while binary size increases remain consistent with the insertion of markers, redirections, and trampolines. In addition, the attack surface analysis with ROPgadget indicates that the effective surface of policy-valid indirect transfers is reduced to less than 1% of the syntactic gadget space identified. These results show that automated static binary rewriting is a viable path for making legacy software compatible with IBT requirements, while preserving executable functionality and supporting future binary-hardening extensions.

Keywords: static binary rewriting, Intel CET, IBT, legacy binaries, control-flow integrity, `endbr64`, `notrack`.

LISTA DE FIGURAS

Figura 1.1 Linha do tempo da “corrida armamentista” entre ataques e defesas.....	14
Figura 2.1 Contraste entre layout normal e após buffer overflow (ilustração do desvio via retorno).	20
Figura 2.2 Exemplo de um <i>exploit</i> ROP (Carlini; Wagner, 2014).	22
Figura 2.3 Controle de fluxo em um exemplo de <i>exploit</i> JOP (Bletsch <i>et al.</i> , 2011).	24
Figura 4.1 Fluxograma da metodologia com as etapas da reescrita binária estática.	48
Figura 5.1 Proporção, em %, de funções protegidas e desprotegidas no nível de otimização -O2 para todos os 102 programas do Coreutils.	60
Figura 5.2 Proporção, em %, de funções protegidas e desprotegidas no nível de otimização -O3 para todos os 102 programas do Coreutils.	61
Figura 5.3 Correlação entre as entradas de função recuperadas e os desvios indiretos identificados no cenário -O0.....	63
Figura 5.4 Distribuição do tamanho binário para os 102 programas do Coreutils em -O0.....	69
Figura 5.5 Distribuição do tamanho binário para os 102 programas do Coreutils em -O2.....	70
Figura 5.6 Distribuição do tamanho binário para os 102 programas do Coreutils em -O3.....	71
Figura 5.7 Distribuição de entradas de função para cada nível de otimização.....	72
Figura 5.8 Distribuição de desvios indiretos para cada nível de otimização.....	73
Figura 5.9 Distribuição de <i>gadgets</i> identificados pelo ROPgadget por nível de otimização. ..	74
Figura 5.10 Comparação agregada entre o espaço sintático de <i>gadgets</i> e os alvos protegidos preservados pela política de instrumentação (em escala logarítmica).....	75

LISTA DE TABELAS

Tabela 2.1 Comparação entre IBT e <i>Shadow Stack</i>	29
Tabela 2.2 Comparativo entre abordagens de reescrita binária.	31
Tabela 3.1 Trabalhos relacionados categorizados.	42
Tabela 3.2 Comparação entre trabalhos relacionados a CET/IBT e a proposta desta dissertação.	44
Tabela 5.1 Overhead médio ponderado, baseado no número de instruções executadas por programa, para contadores do perf.	65
Tabela 5.2 Overhead médio de desempenho por nível de otimização, com desvio padrão, intervalo de confiança de 95% e valores mínimo e máximo entre programas.	66
Tabela A.1 Overhead de desempenho por programa para cada nível de otimização.	88

LISTA DE ABREVIATURAS E SIGLAS

ASLR – Address Space Layout Randomization
BHB – Branch History Buffer
BHI – Branch History Injection
BTB – Branch Target Buffer
BTI – Branch Target Injection
CET – Control-flow Enforcement Technology
CFI – Control-Flow Integrity
CFG – Control-Flow Graph
COP – Call-Oriented Programming
CPU – Central Processing Unit
CSV – Comma-Separated Values
DEP – Data Execution Prevention
ELF – Executable and Linkable Format
GCC – GNU Compiler Collection
IBT – Indirect Branch Tracking
IPC – Instructions per Cycle
IR – Intermediate Representation
JOP – Jump-Oriented Programming
LIEF – Library to Instrument Executable Formats
LLVM – Low Level Virtual Machine
PIE – Position Independent Executable
PMU – Performance Monitoring Unit
ROP – Return-Oriented Programming
RIP – Register Instruction Pointer
RSB – Return Stack Buffer
SLH – Speculative Load Hardening
SPEC – Standard Performance Evaluation Corporation
TVA – Time Variance Authority
VSA – Value-Set Analysis

SUMÁRIO

1	INTRODUÇÃO	13
1.1	CONTEXTUALIZAÇÃO E MOTIVAÇÃO.....	13
1.2	OBJETIVOS	16
1.3	QUESTÕES DE PESQUISA	16
1.4	CONTRIBUIÇÕES.....	17
1.5	ORGANIZAÇÃO DO DOCUMENTO.....	17
2	FUNDAMENTAÇÃO TEÓRICA.....	19
2.1	MÉTODOS DE ATAQUE	19
2.1.1	EXPLORAÇÃO POR ESTOURO DE BUFFER NA PILHA	19
2.1.2	EXPLORAÇÃO POR RETURN-TO-LIBC	20
2.1.3	EXPLORAÇÃO POR <i>RETURN-ORIENTED PROGRAMMING</i>	21
2.1.4	EXPLORAÇÃO POR <i>CALL-ORIENTED PROGRAMMING</i>	22
2.1.5	EXPLORAÇÃO POR <i>JUMP-ORIENTED PROGRAMMING</i>	23
2.2	MECANISMOS DE PROTEÇÃO.....	24
2.2.1	PREVENÇÃO DE EXECUÇÃO DE DADOS (DEP)	24
2.2.2	ESPAÇO ALEATÓRIO DE ENDEREÇOS (ASLR)	25
2.2.3	INTEGRIDADE DO FLUXO DE CONTROLE	26
2.2.4	RASTREAMENTO INDIRETO DE RAMIFICAÇÃO	26
2.2.5	PILHA SOMBRA.....	28
2.2.6	TECNOLOGIA DE PROTEÇÃO DE FLUXO DE CONTROLE	29
2.3	REESCRITA DE BINÁRIOS.....	30
2.4	CONSIDERAÇÕES FINAIS	31
3	TRABALHOS RELACIONADOS	33
3.1	REESCRITA ESTÁTICA COM BAIXA DEPENDÊNCIA DE CFG	33
3.2	CET/IBT E REFINAMENTOS DE GRANULARIDADE FINA.....	35
3.3	RESOLUÇÃO DE SALTOS INDIRETOS.....	35
3.4	AVALIAÇÕES EMPÍRICAS E COMPARATIVOS DE REESCRITA	36
3.5	LIMITES PRÁTICOS DAS DEFESAS BASEADAS EM HARDWARE	37

3.6	CONSIDERAÇÕES FINAIS	39
4	SISTEMA BIN2CET: COMPATIBILIZAÇÃO COM IBT POR REESCRITA BINÁRIA ESTÁTICA	46
4.1	PROBLEMA DE COMPATIBILIZAÇÃO	46
4.2	METODOLOGIA PROPOSTA	47
4.2.1	ANÁLISE	49
4.2.2	GERAÇÃO DE ARTEFATO INTERMEDIÁRIO	50
4.2.3	TRANSFORMAÇÃO	50
4.2.4	SAÍDA E VALIDAÇÃO	52
4.3	CONSIDERAÇÕES FINAIS	53
5	EXPERIMENTOS E RESULTADOS	54
5.1	PROTOCOLO EXPERIMENTAL	54
5.1.1	AMBIENTE E FERRAMENTAS UTILIZADAS	55
5.1.2	PROCEDIMENTO DE MEDIÇÃO E TRATAMENTO ESTATÍSTICO	56
5.1.3	MÉTRICAS	57
5.2	RESULTADOS	58
5.2.1	EFETIVIDADE DA COMPATIBILIZAÇÃO COM IBT	58
5.2.2	RELAÇÃO ENTRE ESTRUTURA DO PROGRAMA E PONTOS DE INSTRUMENTAÇÃO	62
5.2.3	OVERHEAD DE EXECUÇÃO	63
5.2.4	TAMANHO DO BINÁRIO	68
5.2.5	REDUÇÃO DA SUPERFÍCIE DE ATAQUE	71
5.3	CONSIDERAÇÕES FINAIS	76
6	CONCLUSÕES E TRABALHOS FUTUROS	78
6.1	DESAFIOS ENCONTRADOS	81
	REFERÊNCIAS	84
	APÊNDICE A	88

1 Introdução

1.1 Contextualização e Motivação

O cenário contemporâneo de segurança digital tem sido marcado pelo aumento da frequência, da sofisticação e do impacto dos ataques cibernéticos. Segundo levantamento publicado pelo MIT Sloan em 2024, os vazamentos de dados aumentaram aproximadamente 20% em relação ao ano anterior, e muitas organizações passaram a enfrentar múltiplas violações em intervalos relativamente curtos (Stackpole, 2024). Esse cenário evidencia que a proteção de sistemas computacionais não depende apenas da correção de vulnerabilidades pontuais, mas também da adoção de mecanismos capazes de reduzir a exploração de falhas em diferentes camadas de software.

Esses incidentes comprometem informações sensíveis, provocam prejuízos financeiros, geram interrupções operacionais e afetam a reputação de organizações públicas e privadas. No contexto brasileiro, o governo federal registrou quase 58.000 incidentes cibernéticos ou alertas de vulnerabilidade digital nos últimos cinco anos, incluindo mais de 9.000 casos de vazamento de dados em sistemas da União (Caniato, 2025). Esse quadro é especialmente crítico em setores como governo, saúde, finanças e infraestrutura crítica, nos quais ataques podem explorar fragilidades presentes em softwares antigos, sistemas desatualizados e componentes cuja manutenção ou atualização é limitada. Assim, torna-se necessário investigar estratégias de mitigação capazes de proteger sistemas já implantados, inclusive quando o código-fonte não está disponível.

A evolução das técnicas de exploração e das contramedidas associadas revelam uma dinâmica contínua de enfrentamento técnico, marcada pelo avanço sucessivo de ataques e defesas. À medida que novas proteções são incorporadas aos sistemas, técnicas ofensivas mais sofisticadas passam a buscar formas de contorná-las, impulsionando o desenvolvimento de mecanismos de defesa mais robustos. Para ilustrar essa trajetória, a Figura 1.1 apresenta uma linha do tempo com marcos relevantes da evolução de ataques e defesas relacionados à exploração de vulnerabilidades em software.

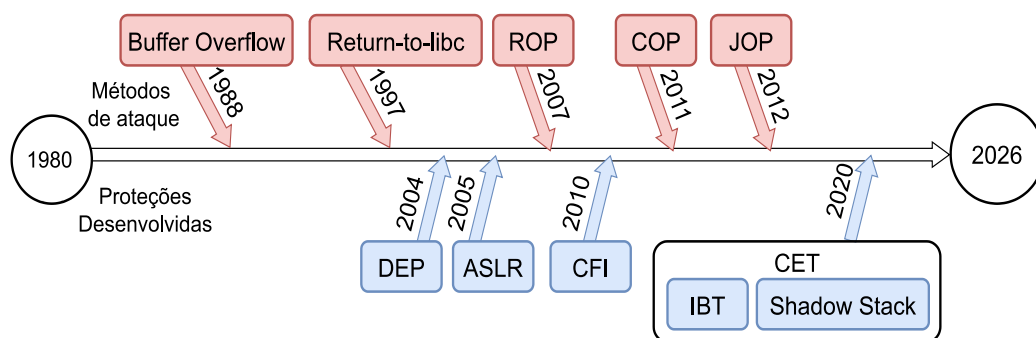


Figura 1.1 Linha do tempo da “corrida armamentista” entre ataques e defesas

Esse panorama histórico revela não apenas o avanço contínuo das técnicas de ataque, mas também o desafio permanente de desenvolver mecanismos capazes de limitar sua efetividade. Ao longo das últimas décadas, foram introduzidas proteções como a Prevenção de Execução de Dados (DEP - *Data Execution Prevention*), a Aleatorização do Layout do Espaço de Endereçamento (ASLR - *Address Space Layout Randomization*) e a Integridade de Fluxo de Controle (CFI - *Control-Flow Integrity*). Essas técnicas elevaram a complexidade da exploração de vulnerabilidades, ao restringir a execução de código injetado, dificultar a previsão de endereços e impor restrições ao fluxo de execução dos programas.

Apesar desses avanços, ataques por reutilização de código, como *Return-Oriented Programming* (ROP) e *Jump-Oriented Programming* (JOP), continuam representando uma ameaça relevante. Essas técnicas exploram vulnerabilidades de corrupção de memória para desviar o fluxo de execução e encadear trechos legítimos de código já existentes no próprio binário, denominados *gadgets*, sem necessariamente injetar novo código executável (Tymburibá *et al.*, 2019). Essa característica permite contornar proteções convencionais, como DEP, e evidencia a necessidade de mecanismos mais robustos para controlar transferências indiretas de execução.

Como resposta a esse cenário, mecanismos de Integridade de Fluxo de Controle passaram a ser propostos para restringir os destinos permitidos de transferências de controle e reduzir a liberdade de redirecionamento explorada por ataques de reutilização de código. Essas ideias foram inicialmente desenvolvidas no contexto de soluções em software, como a formulação clássica de CFI proposta por Abadi *et al.* (2005) e mecanismos de endurecimento explorados em conjuntos de patches como os do PaX Team (2015). Posteriormente, a Intel incorporou parte desses princípios em hardware por meio da *Control-flow Enforcement Technology* (CET), um conjunto de extensões arquiteturais voltado ao fortalecimento da integridade do fluxo de controle. A CET combina dois mecanismos principais: o *Shadow Stack*,

voltado à proteção de endereços de retorno, e o *Indirect Branch Tracking* (IBT), voltado à validação de destinos de chamadas e saltos indiretos. Embora represente um avanço importante na mitigação de desvios maliciosos, sua adoção prática depende da combinação de suporte em hardware, sistema operacional e binários compilados com as marcações esperadas pelo mecanismo. Por isso, uma parcela significativa de softwares legados permanece fora desse modelo de proteção, especialmente quando o código-fonte não está disponível ou quando a recompilação não é viável.

Softwares legados ainda desempenham papéis centrais em setores estratégicos, como saúde, indústria, finanças e infraestrutura crítica. Muitos desses sistemas permanecem em operação por dependerem de ambientes específicos, bibliotecas antigas, requisitos regulatórios ou processos de manutenção consolidados ao longo do tempo. No entanto, por não terem sido projetados para os requisitos atuais de segurança, esses sistemas apresentam dificuldades de compatibilização com mecanismos modernos de defesa. Esse problema é agravado quando o código-fonte original não está disponível, quando a recompilação é inviável ou quando alterações no ambiente de execução podem comprometer a estabilidade do sistema.

Diante dessa limitação, a reescrita binária surge como uma alternativa promissora para adaptar executáveis já compilados a novos requisitos de segurança. Essa técnica permite modificar programas diretamente no nível de código de máquina, sem exigir acesso ao código-fonte original (D’Elia *et al.*, 2022; Duck *et al.*, 2020). A reescrita pode ser realizada de forma dinâmica, durante a execução do programa, ou de forma estática, antes da execução, diretamente sobre o binário em disco. Neste trabalho, adota-se a abordagem estática, pois ela permite produzir executáveis modificados de forma permanente, reduzindo a dependência de instrumentação em tempo de execução e favorecendo a compatibilização prévia com mecanismos como o IBT (INTEL, 2024; Wenzl *et al.*, 2019).

Com base nessa perspectiva, este trabalho propõe o BIN2CET, um sistema automatizado de reescrita binária estática voltada à compatibilização de binários ELF legados com o mecanismo *Indirect Branch Tracking* (IBT) da Intel CET. A proposta organiza o processo de compatibilização como um fluxo de análise, transformação e validação de executáveis, combinando manipulação em nível de ELF, inserção de instruções *endbr64*, uso de trampolins para preservação semântica e aplicação conservadora de *notrack* quando a reescrita segura de determinados destinos não é possível. Dessa forma, o BIN2CET busca viabilizar a adoção parcial de requisitos do IBT em binários legados, sem exigir recompilação, sem depender da recuperação completa do grafo de fluxo de controle e preservando a funcionalidade original do executável.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver e avaliar um sistema automatizado de reescrita binária estática para compatibilizar binários ELF legados com o mecanismo IBT da Intel CET, sem exigir acesso ao código-fonte nem recuperação completa do grafo de fluxo de controle, preservando a funcionalidade original e reduzindo a superfície efetiva de destinos válidos para transferências indiretas.

Para atingir o objetivo geral, propõem-se os seguintes objetivos específicos:

1. Projetar uma estratégia de instrumentação binária que insira instruções *endbr64* em alvos recuperados de desvios indiretos, quando a reescrita do alvo puder ser realizada com segurança.
2. Implementar uma ferramenta de reescrita estática que concretize essa estratégia com microtransformações locais e trampolins, evitando dependência de recuperação completa do grafo de controle de fluxo.
3. Avaliar a eficácia da instrumentação, seus *overheads* de desempenho (tempo de execução, tamanho do binário, consumo de memória) em diferentes contextos bem como a redução da superfície efetiva de ataque com ênfase em softwares legados.

1.3 Questões de Pesquisa

À luz dos objetivos definidos, este trabalho busca responder às seguintes questões de pesquisa:

- Q1. Como compatibilizar binários ELF legados com os requisitos estruturais do mecanismo IBT da Intel CET sem recompilação do código-fonte?
- Q2. Em que medida uma abordagem de reescrita binária estática baseada em microtransformações locais, trampolins e uso conservador de *notrack* preserva a funcionalidade dos executáveis e introduz overheads aceitáveis de desempenho e tamanho binário?
- Q3. Em que medida o BIN2CET reduz a superfície efetiva de ataque associada a transferências indiretas válidas pela política do IBT em binários legados instrumentados?

1.4 Contribuições

As contribuições deste trabalho são:

1. Proposição de uma abordagem de reescrita binária estática para compatibilização de binários ELF legados com o mecanismo IBT da Intel CET, sem exigir acesso ao código-fonte nem recuperação completa do grafo de fluxo de controle.
2. Desenvolvimento do BIN2CET, um sistema automatizado que organiza o processo de compatibilização em um fluxo de análise, transformação e validação de executáveis, combinando inserção de instruções *endbr64*, uso de trampolins para preservação semântica e aplicação conservadora de *notrack* quando a reescrita segura do alvo não é possível.
3. Implementação de microtransformações locais para preservar a funcionalidade de instruções sobrescritas durante a reescrita, incluindo o tratamento de instruções sensíveis à posição, como referências relativas a RIP.
4. Avaliação empírica do sistema em programas da suíte Coreutils, considerando cobertura da instrumentação, preservação funcional, overhead de desempenho, crescimento do tamanho binário e redução da superfície efetiva de ataque associada a transferências indiretas válidas pela política do IBT.
5. Disponibilização de documentação técnica e procedimentos de reprodução para apoiar a execução dos experimentos, a extensão do sistema e a comparação com abordagens futuras de endurecimento binário.

1.5 Organização do Documento

A organização do texto é detalhada a seguir:

- O Capítulo 2 apresenta a fundamentação teórica necessária ao desenvolvimento da pesquisa, incluindo ataques por reutilização de código, mecanismos de proteção em tempo de execução, modelos de fluxo de controle, formatos executáveis e os principais componentes da Intel CET, com ênfase no mecanismo IBT e nos requisitos associados ao uso de instruções *endbr64*.

- O Capítulo 3 discute os trabalhos relacionados em reescrita binária, instrumentação estática e dinâmica, recuperação de fluxo de controle e técnicas de mitigação baseadas em integridade de fluxo de controle. O capítulo posiciona o BIN2CET em relação às abordagens existentes e destaca as lacunas exploradas por este trabalho.
- O Capítulo 4 apresenta o sistema BIN2CET, descrevendo sua arquitetura, seu fluxo de análise, os artefatos produzidos e as estratégias de transformação binária adotadas. Também são detalhados o uso da biblioteca LIEF, a integração com uma versão estendida do E9Patch, a inserção de instruções *endbr64*, o emprego de trampolins, o tratamento de instruções dependentes de RIP e o uso de *notrack* como mecanismo conservador de compatibilidade.
- O Capítulo 5 descreve a metodologia experimental e apresenta os resultados obtidos. São detalhados o corpus baseado na suíte GNU Coreutils, o ambiente de execução, as métricas avaliadas, o procedimento de medição, o tratamento estatístico e os resultados relacionados à cobertura da instrumentação, à preservação funcional, ao desempenho, ao crescimento do tamanho binário e à redução da superfície efetiva de ataque.
- Por fim, o Capítulo 6 apresenta as conclusões do trabalho, sintetizando as contribuições, os principais resultados, as limitações observadas, as ameaças à validade e as direções para trabalhos futuros.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos necessários para compreender a evolução das técnicas de exploração, os principais mecanismos de proteção e os fundamentos da reescrita binária. Para contextualizar o tema, inicia-se pelas técnicas clássicas, com destaque para o estouro de buffer na pilha (*stack buffer overflow*), que serviram de base para explorações modernas. Em seguida, discutimos defesas como a Prevenção de Execução de Dados (DEP, *Data Execution Prevention*), a Aleatorização do Espaço de Endereçamento (ASLR, *Address Space Layout Randomization*) e a Integridade do Fluxo de Controle (CFI, *Control-Flow Integrity*), além de mecanismos de hardware como o IBT e a Pilha Sombra (*Shadow Stack*), pilares da Tecnologia de Aplicação do Fluxo de Controle (CET, *Control-flow Enforcement Technology*) da Intel.

2.1 Métodos de Ataque

2.1.1 Exploração por Estouro de Buffer na Pilha

A pilha (*stack*) é a estrutura de dados usada pelo programa para armazenar variáveis locais, parâmetros e endereços de retorno de funções. Em arquiteturas como x86/x86-64, chamadas (*call*) de função empilham (*push*) o endereço de retorno, enquanto a instrução de retorno (*ret*) restaura o fluxo de execução a partir desse valor. O estouro de buffer ocorre quando operações de escrita ultrapassam o limite de uma área alocada, sobrescrevendo regiões adjacentes da memória, inclusive o endereço de retorno. Com isso, um atacante pode desviar o fluxo de execução para um endereço por ele escolhido. Um dos primeiros marcos históricos associados a essa classe de vulnerabilidades foi o Morris Worm, em 1988 (Orman, 2003), que explorou falhas em larga escala e comprometeu diversas máquinas conectadas à Internet. Posteriormente, a técnica de exploração de estouro de buffer na pilha foi sistematizada e difundida tecnicamente por Aleph One no texto “Smashing the Stack for Fun and Profit” (1996).

Embora antiga, a vulnerabilidade segue relevante, gerando variações e exigindo atualizações constantes de software e de mecanismos de proteção (HP, 2024). A Figura 2.1 ilustra o contraste entre um layout normal e a condição após o overflow.

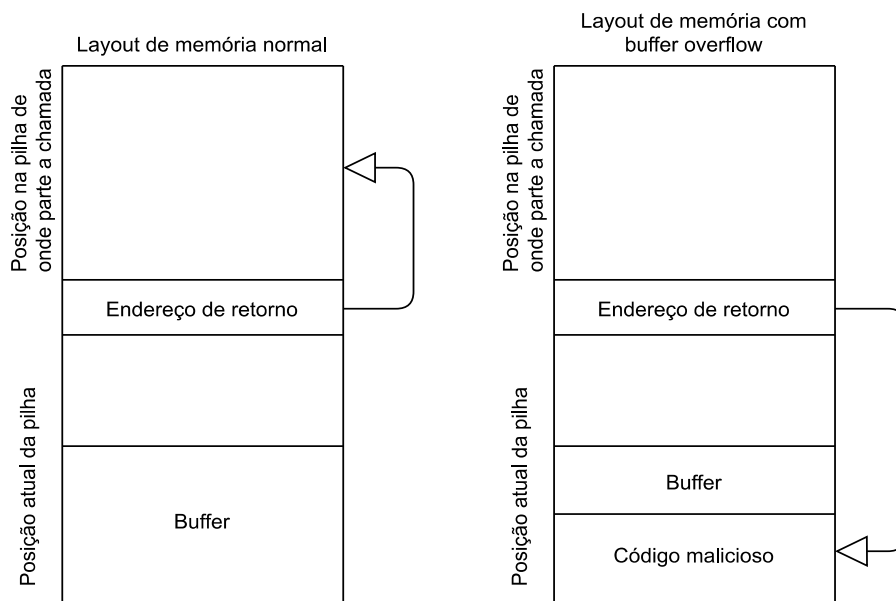


Figura 2.1 Contraste entre layout normal e após buffer overflow (ilustração do desvio via retorno).

2.1.2 Exploração por return-to-*libc*

A *return-into-libc* é uma técnica de exploração que tira proveito de vulnerabilidades no gerenciamento de memória de programas, permitindo que atacantes desviem o fluxo de execução sem a necessidade de injetar código malicioso diretamente no espaço de memória da aplicação. Como apresentado por Shacham (2007), embora a técnica se aproveite de um estouro de *buffer*, em vez de injetar *shellcode*, o atacante redireciona a execução para funções já presentes nas bibliotecas do sistema, como a *libc* no caso dos sistemas Unix-like, para realizar ações arbitrárias.

O atacante desvia a execução usando funções nativas da biblioteca *libc*, como *system()* ou *execve()*. Assim, ele não precisa injetar um código externo (*shellcode*). O ataque usa as próprias funcionalidades normais do sistema para executar comandos maliciosos. A relevância dessa técnica advém do fato de que, ao contrário de outros ataques, ela pode contornar várias proteções de segurança, como a *Data Execution Prevention* (DEP), que impede a execução de código na pilha.

2.1.3 Exploração por *Return-Oriented Programming*

O *Return-Oriented Programming* (ROP) é uma técnica avançada de exploração que permite a execução de código malicioso sem a necessidade de injetar código diretamente no programa. Essa abordagem tira proveito de trechos de código legítimos, chamados *gadgets*, que já estão presentes no programa ou em suas bibliotecas carregadas.

Um gadget é um pequeno bloco de código que normalmente termina com uma instrução de controle de fluxo, como RET, CALL ou JMP. Os atacantes encadeiam esses *gadgets* de forma a construir um fluxo de controle arbitrário, capaz de realizar ações complexas, como chamadas ao sistema ou manipulação direta de memória (Carlini; Wagner, 2014; Tymburibá *et al.*, 2019).

O funcionamento básico do ROP envolve:

1. Identificação de *gadgets*:

O atacante analisa o programa ou suas bibliotecas para encontrar *gadgets* adequados, que podem ser usados para construir a funcionalidade desejada.

2. Manipulação da pilha:

A pilha de execução é alterada pelo invasor para redirecionar o fluxo de controle para os *gadgets* selecionados. Isso é feito modificando os endereços de retorno ou registros usados pelas instruções de controle de fluxo.

3. Encadeamento de *gadgets*:

Os *gadgets* são executados em sequência, permitindo que o atacante controle completamente o fluxo de execução do programa.

O ROP é particularmente perigoso porque enfraquece a eficácia de mecanismos tradicionais quando combinado com condições auxiliares de exploração. Em relação à DEP, o ROP não precisa injetar novo código executável, pois reutiliza instruções já presentes no programa ou em bibliotecas carregadas. Assim, a proteção contra execução de dados deixa de ser suficiente para impedir a execução de uma cadeia maliciosa. Em relação ao ASLR, o ROP não o supera automaticamente. A aleatorização dificulta a exploração ao tornar imprevisíveis os endereços dos gadgets, especialmente quando bibliotecas, pilha, heap e regiões de código são carregadas em posições distintas a cada execução. No entanto, quando o atacante dispõe de

vazamentos de informação, baixa entropia, endereços parcialmente previsíveis ou bibliotecas carregadas em posições conhecidas, torna-se possível localizar *gadgets* e construir cadeias de reutilização de código apesar da aleatorização.

Na Figura 2.2 é possível observar um exemplo de encadeamento de *gadgets*, onde o atacante constrói um fluxo de controle arbitrário para executar comandos maliciosos sem adicionar novo código ao programa.

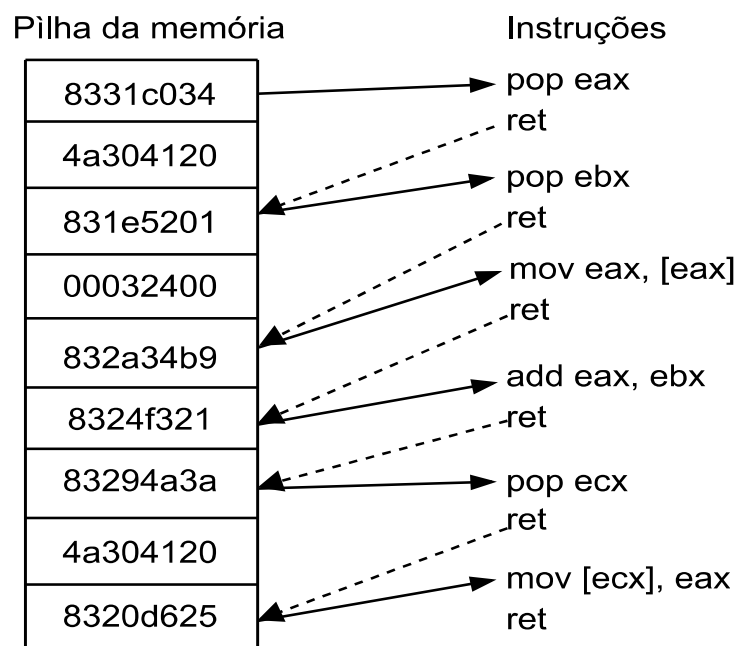


Figura 2.2 Exemplo de um *exploit* ROP (Carlini; Wagner, 2014).

2.1.4 Exploração por *Call-Oriented Programming*

O *Call-Oriented Programming* (COP) é uma variação mais avançada do *Return-Oriented Programming* (ROP). Enquanto o ROP utiliza *gadgets* que terminam em instruções de retorno (RET), o COP expande a exploração para incluir outros tipos de instruções de controle de fluxo, como chamadas de função (CALL) e saltos (JMP).

Os passos para funcionamento do COP envolvem:

1. O atacante identifica trechos de código (*gadgets*) que utilizam chamadas de função ou saltos.

2. Esses *gadgets* são encadeados para realizar ações maliciosas, manipulando a memória ou registros usados nas instruções de controle.
3. O fluxo de controle é redirecionado de forma arbitrária, permitindo a execução de comandos não autorizados.

Embora o COP seja mais difícil de implementar do que o ROP, ele é eficaz em explorar falhas no controle de fluxo, especialmente quando:

- O CFI é mal configurado: Um CFI que não verifica todas as instruções de controle pode permitir o uso de *gadgets* baseados em CALL ou JMP.
- O DEP e ASLR estão ativos: O COP, assim como o ROP, é capaz de contornar as proteções do DEP ao reutilizar código legítimo já presente no programa. O ASLR, por sua vez, dificulta a construção da cadeia de *gadgets* ao aleatorizar endereços, mas pode ser enfraquecido quando o atacante dispõe de vazamentos de informação ou de regiões parcialmente previsíveis.

2.1.5 Exploração por *Jump-Oriented Programming*

O *Jump-Oriented Programming* (JOP) é uma técnica avançada de exploração que, assim como o *Return-Oriented Programming* (ROP) e o *Call-Oriented Programming* (COP), permite que atacantes redirecionem o fluxo de execução de um programa para realizar ações maliciosas. No entanto, o JOP utiliza *gadgets* baseados em instruções de salto (JMP), em vez de retornos (RET) ou chamadas de função (CALL) (Bletsch *et al.*, 2011).

Os passos básicos do JOP incluem:

1. Identificação de *gadgets*:

O atacante analisa o programa para localizar pequenos trechos de código (*gadgets*) que terminam em instruções de salto (JMP) ou saltos condicionais.
2. Manipulação de registros ou memória:

Em vez de modificar a pilha de retorno, como no ROP, o JOP altera registros ou áreas específicas da memória para redirecionar os saltos para *gadgets* escolhidos.
3. Encadeamento dos *gadgets*:
 - Os *gadgets* são organizados de forma que o fluxo de execução siga uma sequência controlada, permitindo que o atacante realize operações arbitrárias, como leitura/escrita em memória ou chamadas ao sistema.

- O JOP pode ser mais difícil de detectar, pois muitas vezes envolve manipulação direta dos saltos dentro do código legítimo, sem a necessidade de modificar o código do programa, como pode ser visto na Figura 2.3.

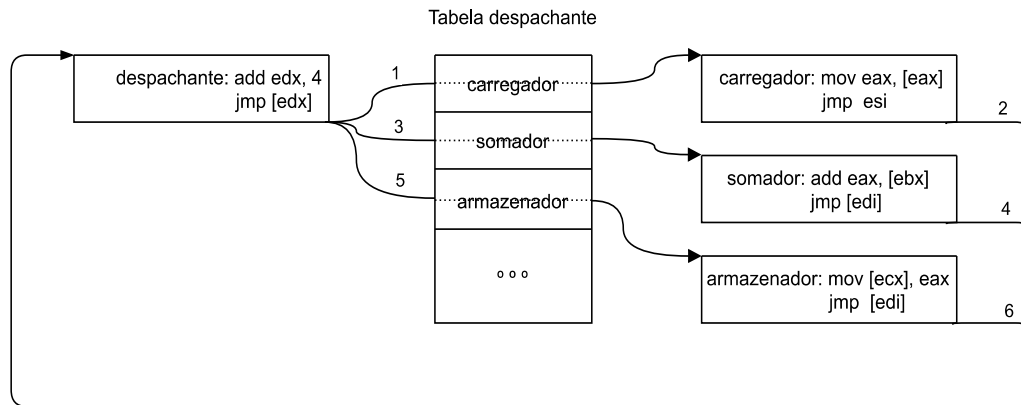


Figura 2.3 Controle de fluxo em um exemplo de *exploit* JOP (Bletsch *et al.*, 2011).

2.2 Mecanismos de Proteção

Esta seção apresenta as defesas em camadas: DEP e ASLR (mitigações de memória), CFI (políticas de controle de fluxo) e os mecanismos de hardware IBT e *Shadow Stack*, integrados na CET. Em cada caso, indicamos o que protegem, como operam, custos/limitações e como se complementam, preparando a discussão sobre compatibilização via reescrita estática para IBT.

2.2.1 Prevenção de Execução de Dados (DEP)

A *Data Execution Prevention* (DEP) é uma medida de segurança introduzida em 2004 nos sistemas operacionais Windows para prevenir a execução de código malicioso em áreas da memória que deveriam ser apenas para dados. A proteção DEP visa bloquear tentativas de execução de código em regiões da memória destinadas apenas a armazenar dados, como a pilha e o *heap*. O princípio do DEP baseia-se na separação de funções da memória: garantir que regiões alocadas para armazenar dados nunca possam ser usadas para executar código. Dessa forma, mesmo que um atacante consiga injetar código malicioso em uma dessas áreas, o sistema bloqueará sua execução, interrompendo o ataque antes que ele cause danos.

Essa proteção funciona ao marcar páginas de memória como "não executáveis". Quando o sistema tenta executar qualquer instrução presente nessas áreas, uma exceção é gerada, impedindo a continuidade do ataque. O DEP pode ser implementado de duas maneiras:

- Software: A proteção é gerenciada pelo sistema operacional.
- Hardware: A funcionalidade é integrada diretamente ao processador, como no suporte à tecnologia NX bit (No-eXecute bit) disponível em processadores modernos.

O DEP passou por diversos aprimoramentos desde sua introdução, tornando-se uma camada fundamental na segurança de sistemas operacionais modernos. No entanto, para que funcione em sua capacidade total, é necessário que o processador e o sistema operacional tenham suporte nativo à tecnologia.

2.2.2 Espaço Aleatório de Endereços (ASLR)

Dentro das estratégias de mitigação desenvolvidas contra a exploração por estouro de *buffer*, foram desenvolvidos mecanismos de proteção de memória, como é o caso do *Address Space Layout Randomization* (ASLR), que foca em tentar evitar que um atacante descubra para qual endereço o fluxo de execução do programa será transferido.

Seu funcionamento faz com que os endereços de memória atribuídos na segmentação do programa sejam atribuídos de forma aleatória para que o atacante não consiga prever qual endereço ele deve inserir seu código malicioso. A implementação do ASLR varia entre sistemas operacionais:

1. Linux:

Adotado a partir do kernel 2.6.12, o ASLR no Linux realiza a aleatorização dos segmentos de memória, incluindo endereços base do código, do *heap*, da pilha e das bibliotecas carregadas. Essa distribuição ocorre cada vez que um processo é iniciado, aumentando significativamente a complexidade de ataques baseados em memória.

2. Windows:

Introduzido no Windows Vista e Windows Server 2008, o ASLR realiza a aleatorização de endereços a cada carregamento de processo. Isso inclui bibliotecas dinâmicas (DLLs), a pilha e o *heap*, garantindo que cada execução do programa utilize um layout de memória diferente, dificultando a previsibilidade de ataques (Hawkins *et al.*, 2017).

Essa técnica, embora não elimine completamente as vulnerabilidades de memória, reduz consideravelmente a probabilidade de sucesso de um ataque, forçando os invasores a recorrerem a métodos mais sofisticados, como a exploração de informações vazadas.

2.2.3 Integridade do Fluxo de Controle

A Integridade do Fluxo de Controle, do inglês *Control Flow Integrity* (CFI), é uma técnica de segurança concebida para garantir que a execução de um programa permaneça dentro de um fluxo de controle legítimo. Seu princípio fundamental é restringir as instruções de desvio (como chamadas de função, retornos e saltos indiretos) a destinos previamente autorizados, prevenindo alterações maliciosas na lógica do programa. Essa abordagem se mostra particularmente eficaz contra técnicas como o *Return-Oriented Programming* (ROP), cuja estratégia consiste em redirecionar a execução para pequenos trechos de código legítimo (*gadgets*) com o objetivo de executar ações arbitrárias.

Diferentemente de mecanismos como DEP e ASLR, que atuam sobre o uso e a organização da memória, o CFI atua diretamente sobre o grau de liberdade do fluxo de execução. Ele define políticas que especificam quais transições de controle são permitidas, como, por exemplo, quais alvos podem receber saltos indiretos ou de quais pontos uma função pode retornar. Essas políticas podem variar em granularidade: políticas mais finas, como por-função ou por-grafo, oferecem maior precisão e, conseqüentemente, maior proteção; por outro lado, políticas mais amplas, com menor detalhamento, implicam em menor sobrecarga computacional, mas deixam espaço para explorações como ROP, COP e JOP.

A implementação prática do CFI geralmente requer instrumentação do binário (seja no momento da compilação ou por meio de técnicas de reescrita) além de suporte em tempo de execução para a verificação das transições de controle. A eficácia da abordagem, portanto, depende diretamente da precisão da política adotada e da cobertura dos pontos de verificação inseridos. Políticas imprecisas ou parcialmente aplicadas tendem a ser insuficientes diante de ataques mais sofisticados, tornando essencial o equilíbrio entre robustez da proteção e viabilidade de aplicação em diferentes contextos, especialmente em softwares legados.

2.2.4 Rastreamento Indireto de Ramificação

O Rastreamento Indireto de Ramificação, do inglês *Indirect Branch Tracking* (IBT), é o mecanismo da Intel CET (INTEL, 2024) que materializa em hardware uma política de validação de destinos de transferências indiretas de controle. Seu princípio está alinhado à tradição de mecanismos de Integridade de Fluxo de Controle, originalmente explorados em soluções de software, como a proposta de CFI de Abadi *et al.* (2005) e mecanismos de endurecimento do PaX Team (2015). No caso do IBT, essa política é implementada na

arquitetura x86 por meio da exigência de que chamadas e saltos indiretos tenham como destino pontos explicitamente marcados no código. Esse tipo de transferência é comum em programas modernos, por exemplo em chamadas por ponteiros de função, métodos virtuais, *callbacks* e tabelas de despacho. Do ponto de vista de segurança, ele representa um vetor crítico de ataque, pois pode permitir que um explorador redirecione a execução para locais não previstos, incluindo *gadgets* usados em ataques ROP, JOP ou COP.

Para mitigar essa ameaça, o IBT define que todo destino legítimo de uma transferência indireta deve iniciar com uma instrução sentinela especial chamada *endbranch*, representada por *endbr32* em arquiteturas de 32 bits e por *endbr64* em arquiteturas de 64 bits. Durante a execução, quando o processador encontra uma chamada ou salto indireto sob a política de IBT, o endereço de destino deve corresponder a um ponto de entrada explicitamente marcado. Caso o destino não inicie com a instrução esperada, a execução pode ser interrompida por uma exceção de proteção de controle. Com isso, o processador restringe desvios para posições arbitrárias dentro do código, especialmente para instruções intermediárias ou regiões que não foram previamente marcadas como destinos válidos.

A instrumentação de executáveis para compatibilidade com IBT exige, portanto, que os destinos válidos de transferências indiretas sejam identificados e anotados com *endbr64*. No entanto, essa inserção não é trivial em binários já compilados. Como o código possui um layout fixo, a introdução da instrução *endbr64* de 4 bytes pode sobrescrever instruções existentes, violar fronteiras de instrução, alterar alinhamentos ou invalidar operandos dependentes da posição da instrução. Esse último caso é particularmente relevante em instruções que utilizam endereçamento relativo a RIP, isto é, relativo ao registrador de ponteiro de instruções da arquitetura x86-64, responsável por indicar a posição da próxima instrução a ser executada. Nesse modo de endereçamento, o endereço efetivo acessado pela instrução é calculado a partir da posição atual do código. Assim, quando uma instrução desse tipo é movida para outra região do binário, como um trampolim, seus deslocamentos precisam ser recalculados para que ela continue acessando o mesmo alvo semântico do programa original.

Além da inserção de *endbr64* nos destinos, a arquitetura x86 também permite o uso do prefixo *notrack* em determinadas transferências indiretas. Esse prefixo desabilita o rastreamento de destino para aquela transferência específica. Do ponto de vista de compatibilidade, *notrack* é útil quando o destino não pode ser reescrito com segurança ou quando a análise não consegue normalizar completamente aquele caso. Entretanto, essa estratégia é menos restritiva do ponto de vista de segurança, pois contorna a checagem do alvo pelo IBT naquela transferência. Assim, a estratégia adotada nesta dissertação prioriza a

instrumentação de destinos com *endbr64* e utiliza *notrack* apenas de forma conservadora, como mecanismo de *fallback* quando a reescrita segura do alvo não é possível.

Nesse cenário, a ausência dos marcadores exigidos pelo IBT impede a execução segura de binários legados em ambientes que impõem essa proteção. A abordagem proposta nesta dissertação contorna essa limitação por meio de reescrita binária estática, buscando inserir *endbr64* em alvos recuperados de desvios indiretos, preservar instruções sobrescritas com trampolins cientes de realocação e empregar *notrack* apenas nos casos em que a normalização do destino não pode ser realizada com segurança. Dessa forma, a compatibilização com IBT é tratada como um problema de transformação local e preservação semântica, sem exigir recompilação do código-fonte nem recuperação perfeita do grafo de fluxo de controle.

2.2.5 Pilha Sombra

A Pilha Sombra, do inglês *Shadow Stack*, é uma técnica de segurança destinada à proteção dos endereços de retorno armazenados na pilha de chamadas durante a execução de um programa. Em um funcionamento tradicional, esses endereços permanecem vulneráveis à corrupção, o que pode ser explorado por atacantes para desviar o fluxo de execução por meio de ataques que substituem o valor legítimo de retorno.

O mecanismo da Pilha Sombra resolve essa fragilidade mantendo uma cópia protegida da pilha de retorno, isolada da pilha convencional do programa. A cada instrução de chamada (CALL), o endereço de retorno é armazenado simultaneamente nas duas pilhas, e, ao atingir uma instrução de retorno (RET), o processador ou o mecanismo de proteção compara ambos os valores. Caso uma divergência seja identificada, a execução é interrompida, bloqueando a tentativa de desvio e impedindo que a manipulação maliciosa seja concluída.

A adaptação de binários para incorporar a Pilha Sombra é consideravelmente mais complexa do que a instrumentação necessária para o IBT. Enquanto o IBT demanda apenas a anotação dos destinos de saltos indiretos com instruções sentinela, a *Shadow Stack* requer reescrita profunda e sincronização precisa entre chamadas e retornos, alterando a lógica estrutural do programa. Esse processo impõe desafios adicionais em termos de compatibilidade, manutenção da semântica original e controle da sobrecarga em tempo de execução.

A Tabela 2.1 sintetiza as diferenças centrais entre o IBT e a *Shadow Stack*, destacando como ambos os mecanismos se complementam dentro da arquitetura CET. A comparação evidencia que o IBT atua de forma mais localizada sobre chamadas e saltos indiretos, enquanto a *Shadow Stack* abrange o comportamento de chamadas e retornos ao longo de todo o programa.

Essa diferença realça a delimitação do escopo desta dissertação, pois embora a *Shadow Stack* seja discutida como parte da fundamentação teórica do mecanismo CET, a implementação proposta e a avaliação experimental concentram-se no IBT, por se tratar de um alvo mais viável para compatibilização por reescrita binária estática.

Tabela 2.1 Comparação entre IBT e *Shadow Stack*.

Aspecto	IBT	<i>Shadow Stack</i>
Objetivo	Proteger transferências indiretas de controle (jmp e call indiretos).	Proteger endereços de retorno (CALL, RET).
Impacto no programa	Localizado, afeta apenas saltos indiretos.	Extensivo, afeta todas as interações de chamada e retorno.
Dificuldade de implementação	Relativamente simples, menos invasivo.	Complexo, exige alterações profundas no binário.
Vulnerabilidade restante	Falha ao identificar todos os pontos de salto indireto.	Sincronização incorreta entre pilhas pode quebrar o programa.

2.2.6 Tecnologia de Proteção de Fluxo de Controle

A *Control-flow Enforcement Technology* (CET), disponibilizada pela Intel em processadores comerciais a partir da linha Tiger Lake, é uma tecnologia de segurança avançada, projetada para proteger a integridade do fluxo de controle dos programas em execução. Seu objetivo é mitigar ataques sofisticados que manipulam o fluxo de execução, como o *Return-Oriented Programming* (ROP) e o *Jump-Oriented Programming* (JOP), que exploram instruções de memória para ganhar controle sobre o programa.

A CET combina o IBT e a *Shadow Stack* como mecanismos complementares. O IBT restringe transferências indiretas de controle a destinos explicitamente marcados como válidos, enquanto a *Shadow Stack* protege endereços de retorno contra adulterações na pilha convencional. Em conjunto, esses mecanismos fortalecem a proteção contra ataques que manipulam o fluxo de execução. No entanto, sua adoção em softwares legados apresenta desafios distintos. A compatibilização com *Shadow Stack* exige alterações mais amplas no comportamento de chamadas e retornos, enquanto o IBT permite uma abordagem mais localizada, voltada à identificação e marcação de destinos válidos de chamadas e saltos

indiretos. Por essa razão, essa dissertação concentra-se na compatibilização de binários legados com o mecanismo IBT.

2.3 Reescrita de Binários

A reescrita binária é uma técnica que viabiliza a modificação de programas já compilados, sem a necessidade de acesso ao código-fonte original. Essa abordagem tem se mostrado estratégica para enfrentar desafios persistentes na área de segurança cibernética, especialmente no contexto de programas legados, cuja manutenção ou adaptação se torna inviável pelas vias tradicionais de engenharia. Ao atuar diretamente sobre os binários, essa técnica permite a análise, inserção de mecanismos de proteção, aplicação de correções e a compatibilização com padrões modernos de execução seguros (Carlini; Wagner, 2014; Gaidis *et al.*, 2023), como os definidos pela tecnologia CET.

As ferramentas capazes de realizar reescrita binária precisam ser robustas o suficiente para operar em múltiplas arquiteturas e sistemas operacionais, respeitando as particularidades de cada ambiente e mantendo atenção para que as modificações realizadas sobre o binário preservem a semântica do programa original, evitando falhas em tempo de execução. A instrumentação de executáveis envolve inserir checagens de segurança, desviar rotas de controle e adaptar o código para funcionar com defesas como o DEP ou o CFI (Meng; Liu, 2016).

Essa alteração pode ser realizada por duas abordagens principais: reescrita binária estática e reescrita binária dinâmica. A reescrita binária estática, adotada neste trabalho, é realizada antes da execução do programa, diretamente sobre o binário armazenado em disco. Essa estratégia permite alterações estruturais, como a inserção de instruções sentinela, a criação de trampolins e rotinas auxiliares, além do redirecionamento de saltos (INTEL, 2024). Trata-se de uma abordagem menos intrusiva em tempo de execução e mais apropriada para o tratamento de softwares legados, pois as alterações não dependem de monitoramento contínuo durante a execução.

No contexto deste trabalho, a reescrita estática é empregada para introduzir compatibilidade com IBT em binários ELF legados. Isso envolve mais do que adicionar instruções *endbr64*. A transformação precisa respeitar fronteiras de instrução, preservar o layout executável sempre que possível e manter a semântica das instruções originalmente presentes no ponto reescrito. Quando uma instrução precisa ser sobrescrita para dar lugar a um marcador de entrada ou a um redirecionamento, seu comportamento deve ser preservado em

uma região auxiliar, geralmente implementada por meio de trampolins. Esse cuidado é particularmente importante para instruções dependentes da posição, como acessos relativos a RIP, cujos deslocamentos precisam ser recalculados quando a instrução é movida para outra região do binário.

Em contrapartida, a reescrita dinâmica atua em tempo de execução e depende de ferramentas especializadas capazes de interagir com a memória do processo ativo. Essa abordagem oferece maior flexibilidade e adaptabilidade, especialmente em cenários onde o código se automodifica dinamicamente durante a execução do programa ou utiliza ofuscação (conjunto de técnicas utilizadas para dificultar a compreensão, a análise e a engenharia reversa de um programa). Em contextos legítimos, é empregada para proteger a propriedade intelectual ou evitar adulterações. No entanto, seu custo de implementação é elevado, requer instrumentação contínua e pode introduzir sobrecarga de desempenho considerável, o que a torna menos adequada para aplicações que demandam estabilidade e portabilidade (Shacham, 2007).

A Tabela 2.2 resume comparativamente as duas abordagens, apresentando suas diferenças de forma objetiva.

Tabela 2.2 Comparativo entre abordagens de reescrita binária.

Modelo/Critério	Reescrita Binária Estática	Reescrita Binária Dinâmica
Escopo	Antes da execução	Durante a execução
Desempenho	Baixa sobrecarga	Moderada a alta
Complexidade	Moderada	Alta
Robustez contra código auto-modificável	Baixa	Alta

2.4 Considerações Finais

A análise histórica e técnica das principais técnicas de exploração e mitigação apresentadas neste capítulo forneceu o embasamento necessário para a compreensão da atual paisagem de ameaças e defesas no domínio da segurança binária. Desde os primeiros ataques baseados em estouros de buffer até os modelos mais sofisticados, como ROP, JOP e COP, é evidente a evolução contínua das estratégias adversárias no sentido de contornar as proteções existentes.

Em contrapartida, observa-se o desenvolvimento de mecanismos de defesa cada vez mais robustos, como DEP, ASLR, CFI e, mais recentemente, a *Control-flow Enforcement Technology* (CET), que introduz os componentes IBT e *Shadow Stack* com suporte direto do hardware. Esses mecanismos representam um avanço significativo na consolidação da integridade do fluxo de controle e têm se mostrado eficazes, ainda que apresentem desafios práticos de adoção, especialmente em softwares legados.

Nesse cenário, a reescrita binária desponta como uma alternativa tecnicamente viável para adaptar sistemas legados às exigências contemporâneas de segurança. Sua capacidade de modificar executáveis de forma independente do código-fonte torna essa técnica estratégica tanto para a modernização de sistemas quanto para o fortalecimento da segurança em ambientes com restrições operacionais.

3 Trabalhos Relacionados

O campo de segurança de software avançou além da simples catalogação de vulnerabilidades. Pesquisas em reescrita/instrumentação binária, proteção do fluxo de controle e compatibilização com defesas de hardware ganharam proeminência diante da escalada de ataques por reutilização de código e da dificuldade prática de modificar binários (executáveis). Este capítulo examina criticamente os principais trabalhos que compõem esse cenário, buscando não apenas mapear o estado da arte, mas também evidenciar lacunas e oportunidades que fundamentam a proposta desta dissertação.

Para dar unidade à análise, adotamos um encadeamento que vai da modificação do binário às garantias de execução e suas evidências práticas, concluindo com os limites atuais: i) reescrita estática com baixa dependência de CFG (Seção 3.1), enfatizando abordagens que preservam endereços e evitam remontagens amplas; ii) CET/IBT e refinamentos de granularidade fina (Seção 3.2), explorando garantias em hardware e restrições adicionais por software; iii) resolução de saltos indiretos (Seção 3.3), tratando da precisão estática e alternativas híbridas; iv) avaliações empíricas e comparativos de reescrita (Seção 3.4), com foco em resultados sobre binários reais; e, por fim, v) limites práticos das defesas baseadas em hardware (Seção 3.5).

3.1 Reescrita Estática com Baixa Dependência de CFG

A reescrita binária estática consolidou-se como uma das principais abordagens para instrumentar executáveis sem acesso ao código-fonte, permitindo a inserção de mecanismos de segurança antes da execução. As primeiras gerações, como DIABLO (Van Put *et al.*, 2005), mostraram que é possível reescrever em tempo de edição visando otimização e endurecimento (*hardening*), mas evidenciaram a sensibilidade à simbolização e reconstrução de CFG. Em busca de maior expressividade, surgiram abordagens que elevam o binário a uma Representação Intermediária (IR, Intermediate Representation) ou remontam grandes trechos do executável, como Zipr (Hawkins *et al.*, 2017; Hiser *et al.*, 2017), RetroWrite (Dinesh *et al.*, 2020) e Ramblr (Wang *et al.*, 2017). Essa linha facilita transformações inspiradas em compiladores, como sanitização, adaptação para *fuzzing* e *inlining* controlado, mas reintroduz a dependência de uma reconstrução precisa do CFG sob otimizações agressivas, chamadas finais, *jump tables* compactas e regiões ambíguas entre código e dados. Mais recentemente, Kim *et al.* (2025), ao proporem o SURI, reforçam que a reconstrução correta de binários x86-64 modernos continua

sendo um problema tecnicamente difícil, especialmente pela necessidade de simbolizar referências de memória e preservar a semântica original do programa. Esse resultado atualiza a discussão ao mostrar que, mesmo com avanços na reconstrução de binários, transformações amplas ainda precisam lidar cuidadosamente com instruções de tamanho variável, referências relativas e dependências de posição.

Em contraste, arquiteturas que minimizam a dependência de CFG privilegiam microtransformações locais e trampolins para preservar endereços e reduzir o risco de quebras semânticas. O E9Patch, introduzido por Duck *et al.* (2020), representa esse desenho ao permitir a reescrita binária estática sem depender da recuperação completa do CFG. Em vez de levantar o binário para uma IR de alto nível, o E9Patch aplica transformações pontuais por meio de trampolins, aproveitamento de bytes de preenchimento e realocação controlada de instruções, preservando os alvos originais dos saltos sempre que possível. Essa estratégia é mais robusta para binários grandes e complexos, nos quais heurísticas de CFG podem falhar.

Para viabilizar a instrumentação segura sem a reconstrução completa do fluxo de controle, o E9Patch utiliza técnicas como *instruction punning* (substituição de instruções por saltos com codificação de mesmo tamanho), *padding reuse* (ocupação de regiões de preenchimento) e *eviction* (realocação de instruções em trampolins auxiliares). Esses mecanismos preservam os alvos dos saltos indiretos originais, assegurando que a semântica do programa não seja comprometida, mesmo em cenários com presença de instruções sobrepostas ou fluxos indiretos não triviais.

Essa abordagem confere ao E9Patch uma escalabilidade superior em relação a reescritores que requerem IRs de alto nível, especialmente ao lidar com executáveis amplos e complexos como navegadores. Avaliações comparativas confirmam sua robustez em tarefas simples de instrumentação, sendo um dos poucos reescritores capazes de operar com sucesso em 100% dos binários testados em contextos reais.

A proposta apresentada por Duck *et al.* (2020) fundamenta parte importante da metodologia adotada nesta dissertação. Mais do que uma simples escolha de ferramenta, sua arquitetura representa uma solução concreta para o desafio central da reescrita segura de binários *stripped*. A impossibilidade de recuperar com precisão o Grafo de Fluxo de Controle (CFG) em binários x86-64 desprovidos de símbolos compromete abordagens que dependem de elevação para representações intermediárias, como aquelas utilizadas por reescritores baseados em LLVM IR. Dessa forma, a arquitetura de instrumentação local e não intrusiva do E9Patch permite a aplicação de modificações estruturais com confiabilidade, mesmo na ausência de

garantias sobre o fluxo de execução original, conferindo à metodologia aqui proposta uma base sólida e operacionalmente viável.

3.2 CET/IBT e Refinamentos de Granularidade Fina

A *Control-flow Enforcement Technology* (CET) integra Rastreamento de Desvios Indiretos (IBT) e Pilha Sombra (*Shadow Stack*) para fortalecer a integridade do fluxo de controle. Em sua forma padrão, o IBT fornece garantia estrutural de granularidade grossa: todo destino de salto indireto deve começar com *endbranch*, o que evita entradas “no meio” de instruções e destinos não autorizados estruturalmente. Contudo, o IBT não distingue contexto: um destino estruturalmente válido pode ser indevido quanto à assinatura/contrato de chamada, por exemplo, permitindo que uma chamada legítima seja redirecionada a um gadget controlado por um atacante.

Para minimizar este problema, Gaidis *et al.* (2023) propõem o FineIBT, uma extensão de software que introduz verificações adicionais para refinar o controle de fluxo. A solução é implementada em nível de compilador (LLVM) e utiliza instrumentação por módulo de tempo de execução (IRM) para garantir que apenas alvos com a aridade (número de argumentos aceitos) e o tipo esperados possam ser alcançados. A abordagem reduz o conjunto de destinos válidos em transferências indiretas e se mostra eficaz contra ataques sofisticados, mantendo uma sobrecarga de desempenho inferior a 2% em benchmarks como SPEC CPU2017.

O FineIBT representa uma inspiração direta para o direcionamento evolutivo desta pesquisa. Embora o presente trabalho tenha como foco inicial a compatibilização binária com o mecanismo IBT da Intel CET, a abordagem delineada por Gaidis *et al.* (2023) ilustra o potencial de aprimoramento das garantias de segurança estrutural mediante verificações complementares de tipo e contexto em desvios indiretos.

A principal limitação do FineIBT, entretanto, reside em sua dependência do código-fonte, o que inviabiliza sua aplicação a binários legados. Esse obstáculo evidencia a relevância de investigações que explorem princípios semelhantes de verificação de tipo e destino, mas operando diretamente sobre o binário, sem recompilação. Abordagens desse tipo ainda permanecem pouco exploradas na literatura.

3.3 Resolução de Saltos Indiretos

A precisão na identificação de destinos de saltos indiretos é determinante para instrumentações corretas e completas. Técnicas de *Value-Set Analysis* (VSA), como no

DEEPVSA (Guo *et al.*, 2019), inferem conjuntos de valores possíveis para registradores/endereço, ampliando a cobertura de destinos potencialmente alcançáveis. Ao estender a VSA com aprendizado de máquina para lidar com ambiguidades e regiões de memória “prováveis”, obtêm-se ganhos de acurácia em cenários difíceis.

O DEEPVSA amplia os limites da VSA tradicional ao incorporar aprendizado de máquina para inferir regiões de memória prováveis em situações ambíguas. A abordagem foi originalmente proposta para análise *post-mortem*, mas ilustra o potencial de métodos híbridos que aliam técnicas estáticas e preditivas para refinar a análise de saltos.

Outra linha recente explora propriedades introduzidas pelo próprio CET para melhorar a precisão da análise estática. Zhao *et al.* (2025) propõem o *Time Variance Authority* (TVA), um framework que utiliza a presença de *endbr64* como indicação de destinos válidos para transferências indiretas. Ao reconhecer que, em um binário compatível com CET, destinos legítimos de desvios indiretos devem iniciar com essa instrução, a abordagem reduz caminhos espúrios durante a desmontagem e melhora a precisão da análise. Esse resultado é relevante para esta dissertação porque evidencia que *endbr64* não atua apenas como marcador de execução para o processador, mas também pode servir como informação estrutural útil para análise e reescrita de binários.

Apesar disso, o custo computacional e a complexidade de engenharia dessas análises tornam-nas desafiadoras como componente operacional em *pipelines* de reescrita ampla. Na prática, observa-se uma tendência a abordagens híbridas: uma análise estática conservadora suficiente para não perder destinos relevantes à tarefa, como os pontos de entrada plausíveis para *endbranch* e, onde persistem ambiguidades, classificadores leves ou priorização para inspeções mais profundas. Esta dissertação adota uma abordagem para garantir a exatidão estrutural mínima exigida pelo IBT, deixando a exploração de híbridos estático–preditivos como trabalho futuro.

3.4 Avaliações Empíricas e Comparativos de Reescrita

Esta seção sintetiza evidências experimentais que comparam reescritores binários em binários reais (frequentemente *stripped*), com o objetivo de fundamentar a opção por microtransformações locais e baixa dependência de IR/CFG. Avaliações abrangentes relatam que abordagens que elevam o binário para uma IR e dependem de reconstrução completa do CFG, como ReOpt, Rev.ng, RetroWrite, tendem a falhar com maior frequência em cenários de

mundo real, enquanto cadeias *disassembly-first* e de transformação local (p.ex., E9Patch, ddisasm) exibem maior confiabilidade operacional em tarefas de instrumentação básicas.

Schulte *et al.* (2022) apresentam uma avaliação abrangente que comparou dez reescritores binários. O estudo demonstrou que ferramentas como ReOpt, Rev.ng e Retrowrite, embora ambiciosas, fracassam em lidar com binários do mundo real com a mesma confiabilidade observada em soluções como o E9Patch e o ddisasm.

Em particular, o E9Patch apresentou taxas de sucesso de 100% na reescrita de binários compilados com GCC para tarefas básicas de instrumentação, evidenciando que estratégias que evitam a análise semântica profunda podem oferecer maior robustez operacional. A análise dos autores reforça a ideia de que, em contextos onde a escalabilidade e a confiabilidade são prioritários, é preferível abdicar da expressividade de transformações em IR em favor de técnicas locais, mais limitadas, porém seguras.

A avaliação comparativa conduzida por Schulte *et al.* (2022) fornece um respaldo empírico valioso para a escolha metodológica feita nesta dissertação. Ao comparar diferentes plataformas de reescrita binária, o estudo demonstra que soluções que evitam a reconstrução completa do controle de fluxo, apresentam maior taxa de sucesso e confiabilidade operacional em binários do mundo real. Essa constatação valida a premissa central do projeto: priorizar a robustez estrutural da instrumentação mesmo em detrimento de transformações mais ambiciosas em representações intermediárias. A evidência de que ferramentas baseadas em IR falham frequentemente ao lidar com executáveis *stripped* reforça a legitimidade técnica da abordagem adotada, consolidando-a como uma escolha fundamentada, confiável e alinhada com os desafios práticos enfrentados na instrumentação de software legado.

3.5 Limites Práticos das Defesas Baseadas em Hardware

A ilusão de invulnerabilidade proporcionada por mecanismos de defesa baseados em hardware, como o CET, é frequentemente desfeita por ataques sofisticados. O estudo técnico conduzido por Malvica (2022), publicado no blog da Offensive Security, demonstra um *bypass* à *Shadow Stack* da CET por meio de uma técnica conhecida como *Counterfeit Object-Oriented Programming* (COOP). O ataque cria objetos falsos em regiões previsíveis da memória e explora chamadas virtuais legítimas para encadear *gadgets* com instruções de controle.

Esse tipo de exploração confirma que, embora o CET represente um avanço significativo na proteção de transferências de controle, sua eficácia depende da integridade estrutural e semântica do binário. Proteções como o IBT, que restringe destinos arquiteturais de

transferências indiretas, também possuem limitações quando não acompanhadas por mecanismos complementares de verificação de tipo e contexto.

Essa limitação também aparece em ataques de execução transiente, especialmente variantes relacionadas ao Spectre v2. Nesse tipo de ataque, o processador pode executar temporariamente instruções em um caminho especulativo incorreto e deixar efeitos observáveis em estruturas microarquiteturais, como caches ou preditores de desvio. Barberis *et al.* (2022), ao introduzirem *Branch History Injection* (BHI), mostram que ataques em cenários de privilégios cruzados podem explorar o histórico global de desvios para influenciar previsões especulativas mesmo em sistemas que adotam mitigadores de hardware projetados para isolar estados de predição entre domínios de privilégio. Embora o BHI não seja um ataque contra o IBT em si, ele demonstra que mecanismos de controle de fluxo arquitetural e mitigadores de predição não eliminam completamente a possibilidade de desvios especulativos exploráveis.

Resultados mais recentes reforçam esse cenário. Wiebing *et al.* (2024), em InSpectre Gadget, analisam a superfície residual de ataques Spectre v2 em cenários de privilégios cruzados e demonstram que ainda existem *gadgets* exploráveis mesmo na presença de mitigadores modernos. Entre esses mitigadores estão o eIBRS, mecanismo da Intel para restringir a influência de previsões de desvio entre contextos de privilégio, restrições ao eBPF, subsistema do kernel Linux usado para carregar programas verificados em tempo de execução, e mecanismos relacionados a IBT e FineIBT. O trabalho também propõe uma ferramenta para inspecionar *gadgets* Spectre com maior precisão, considerando restrições de explorabilidade que ferramentas puramente sintáticas não capturam. Esse resultado é relevante para esta dissertação porque reforça a distinção entre reduzir destinos arquiteturais válidos e eliminar completamente superfícies residuais exploráveis em execução especulativa.

Entre as propostas defensivas recentes, Baumann *et al.* (2026) propõem o TrioSecuris, uma proteção formalmente verificada contra sequestro especulativo de fluxo de controle. A abordagem combina CFI assistido por hardware no estilo CET com *speculative load hardening* (SLH), técnica inserida pelo compilador para reduzir vazamentos causados por execução especulativa. O objetivo é proteger programas contra variantes de Spectre associadas a diferentes estruturas de predição do processador, como o *Branch Target Buffer* (BTB), usado para prever destinos de desvios indiretos, o *Return Stack Buffer* (RSB), usado para prever endereços de retorno, e o *Pattern History Table* (PHT), usado para prever o resultado de desvios condicionais. A existência de uma defesa desse tipo evidencia que mecanismos no estilo IBT fornecem uma base útil, mas precisam ser combinados com proteções adicionais quando o objetivo é mitigar vazamentos decorrentes de execução especulativa.

Esses trabalhos reforçam que não existe uma solução definitiva e isolada para a proteção do fluxo de controle em sistemas modernos. A marcação de destinos válidos por IBT reduz a liberdade de redirecionamento arquitetural, mas não impõe uma política fina de tipo, contexto ou aridade, nem elimina superfícies residuais associadas à execução transiente. Isso é particularmente relevante no caso de binários legados, nos quais não é viável aplicar diretamente técnicas baseadas em compilador ou modificar estruturas internas com suporte explícito.

Este cenário de ameaças em constante evolução justifica a necessidade de investigações contínuas sobre o refinamento dos mecanismos de verificação existentes. Neste contexto, a proposta desta dissertação deve ser compreendida como uma etapa estrutural de compatibilização com IBT por meio de reescrita binária estática. A abordagem busca permitir que binários ELF legados passem a respeitar os requisitos arquiteturais do IBT e reduzam a superfície efetiva de destinos válidos para transferências indiretas. Ao mesmo tempo, os trabalhos recentes discutidos nesta seção mostram que essa compatibilização não elimina a necessidade de mecanismos complementares, como verificações de granularidade fina, mitigadores contra execução especulativa e validações empíricas da superfície residual de ataque.

3.6 Considerações Finais

O levantamento realizado neste capítulo permitiu mapear o estado da arte em frentes diretamente relacionadas à proposta desta dissertação, incluindo reescrita binária estática, compatibilização com mecanismos de proteção de fluxo de controle, análise de binários sem símbolos, avaliação empírica de ferramentas de instrumentação e limites práticos das defesas baseadas em hardware. A organização adotada ao longo das seções teve como objetivo construir uma progressão argumentativa: primeiro foram discutidas técnicas de reescrita e instrumentação; em seguida, mecanismos como CET/IBT e refinamentos de granularidade fina; depois, desafios de análise de saltos indiretos e avaliação de ferramentas; por fim, foram examinadas limitações associadas a defesas de hardware e ataques recentes de execução transiente.

Esse percurso evidencia que a reescrita binária estática permanece uma estratégia relevante para modificar executáveis sem acesso ao código-fonte, mas também mostra que diferentes famílias de abordagens fazem escolhas bastante distintas. Ferramentas baseadas em reconstrução ampla ou representações intermediárias oferecem maior expressividade, porém

dependem de informações estruturais difíceis de recuperar em binários reais. Em contraste, técnicas baseadas em microtransformações locais, trampolins e menor dependência de CFG tendem a oferecer maior robustez operacional, embora com menor liberdade para transformações globais. Essa tensão aparece de forma recorrente na literatura e fundamenta a escolha metodológica desta dissertação.

Os trabalhos recentes reforçam essa leitura. O SURI, ao tratar da reconstrução correta de binários x86-64 modernos, evidencia que a preservação semântica continua sendo um desafio central mesmo em abordagens avançadas de *reassembly*. O TVA, por sua vez, mostra que marcações introduzidas pelo próprio CET, como *endbr64*, podem ser exploradas como informação estrutural para melhorar a desmontagem estática. Já trabalhos como FineIBT demonstram que o IBT pode ser refinado por verificações adicionais de tipo, aridade e contexto, embora essa estratégia dependa de instrumentação em nível de compilador. Esses estudos aproximam-se desta dissertação por tratarem de CET/IBT, análise binária ou controle de fluxo, mas diferem do BIN2CET porque não têm como objetivo principal inserir compatibilidade com IBT em binários ELF legados desprotegidos por meio de reescrita estática local.

A discussão sobre os limites das defesas de hardware também delimita o escopo da contribuição proposta. Trabalhos como *Branch History Injection* e *InSpectre Gadget* mostram que ataques de execução transiente, especialmente variantes de Spectre v2 em cenários de privilégios cruzados, continuam explorando superfícies residuais mesmo na presença de mitigadores modernos e políticas mais restritivas de controle de fluxo, como mecanismos relacionados a IBT e FineIBT. O TrioSecuris, por sua vez, reforça que mecanismos no estilo CET/IBT podem precisar ser combinados com proteções adicionais contra execução especulativa. Assim, a proposta desta dissertação não deve ser interpretada como uma defesa completa contra todas as formas de ataque, mas como uma etapa estrutural para permitir que binários legados passem a respeitar os requisitos arquiteturais do IBT e reduzam a superfície efetiva de destinos válidos para transferências indiretas.

O cenário traçado pelos estudos avaliados reforça a necessidade de abordagens que conciliem precisão analítica, baixo custo computacional e independência de fonte. Tais requisitos orientam os objetivos da presente dissertação e fundamentam a escolha metodológica adotada nos capítulos seguintes.

Para complementar a organização adotada ao longo do capítulo, os trabalhos analisados também foram agrupados em categorias temáticas para fins de comparação nas Tabela 3.1 e Tabela 3.2. Essas categorizações resumem as principais contribuições de cada trabalho e permitem visualizar quais áreas da literatura receberam maior atenção. Apesar de alguns

estudos tratem simultaneamente de reescrita, análise binária, CFI, CET e mitigação de ataques, para a classificação na Tabela 3.1 foi selecionada a categoria predominante de cada artigo.

Categoria 1 – Reescrita binária estática e remontagem de binários sem recompilação: Trabalhos que discutem técnicas de reescrita binária estática voltadas à inserção de proteções, otimizações ou modificações estruturais em executáveis, sem a necessidade de recompilação. Enquadram-se aqui os trabalhos de L. Van Put *et al.* (2005), Smithson *et al.* (2013), Meng e Liu (2016), Hawkins *et al.* (2017), Duck *et al.* (2020) e Kim *et al.* (2025). Esses estudos abordam diferentes formas de reescrita direta, incremental, local ou por remontagem, incluindo desafios como relocação de instruções, preservação de referências dependentes de posição, manutenção do fluxo de controle e integridade semântica do binário.

Categoria 2 – Proteções de fluxo de controle e CET/IBT: Esta categoria reúne trabalhos que discutem o CET da Intel, o IBT, a *Shadow Stack* ou refinamentos de políticas de integridade de fluxo de controle. Shanbhogue *et al.* (2019) descrevem a arquitetura da CET e seus principais mecanismos de proteção. Meng *et al.* (2021) e Xie *et al.* (2022) discutem propostas associadas à adoção ou compatibilização com CET. Gaidis *et al.* (2023), por sua vez, propõem o FineIBT, que adiciona verificações de granularidade fina ao IBT por meio de instrumentação em compilador. Baumann *et al.* (2026) também se enquadram nesta categoria ao propor o TrioSecuris, uma defesa formalmente verificada que combina CFI assistido por hardware no estilo CET com proteção contra execução especulativa. Em conjunto, esses estudos reforçam a importância de mecanismos que validem destinos de transferências indiretas e apontam para a necessidade de políticas complementares ao IBT básico.

Categoria 3 – Avaliação e comparação de ferramentas de instrumentação: Esta categoria reúne trabalhos que investigam, avaliam ou comparam ferramentas e frameworks para reescrita binária, instrumentação ou análise de executáveis. Enquadram-se aqui estudos sobre ferramentas como Uroboros (Wang *et al.*, 2016), Zipr (Hawkins *et al.*, 2017; Hiser *et al.*, 2017) e Dyninst (Bernat, 2011), além de avaliações experimentais conduzidas por Rimsa *et al.* (2019), D’Elia *et al.* (2022) e Schulte *et al.* (2022). Esses trabalhos analisam aspectos como desempenho, cobertura, robustez, artefatos introduzidos e precisão da instrumentação, oferecendo uma base comparativa importante para justificar escolhas metodológicas em cenários com binários reais.

Categoria 4 – Análise binária, desmontagem e binários sem símbolos: Esta categoria concentra trabalhos relacionados aos desafios encontrados com a recuperação de informações estruturais em executáveis, especialmente em binários *stripped* ou com pouca informação

simbólica disponível. Wenzl *et al.* (2019), Bernat e Miller (2011), Zhang *et al.* (2014) e Meng e Miller (2016) discutem desafios como identificação de funções, reconstrução de CFGs, separação entre código e dados e limites da análise estática pura. Zhao *et al.* (2025) também se enquadram nesta categoria ao propor o TVA, que explora marcações *endbr64* introduzidas pelo CET para tornar a desmontagem estática mais precisa e reduzir caminhos espúrios. Esses trabalhos evidenciam que a qualidade da análise binária é um fator decisivo para instrumentações corretas e para a aplicação segura de transformações em executáveis.

Categoria 5 – Limites de hardware, ataques de execução transiente e técnicas de exploração: Esta categoria reúne trabalhos que discutem técnicas de exploração, limitações de defesas baseadas em hardware e superfícies residuais de ataque. Tymburibá *et al.* (2014) propõem um modelo de proteção em múltiplas camadas usando recursos microarquiteturais que contribui para mitigar ataques por reutilização de código, como *Return-Oriented Programming* (ROP). Barberis *et al.* (2022) apresentam o *Branch History Injection*, evidenciando que ataques de execução transiente relacionados ao Spectre v2 podem explorar o histórico de desvios mesmo diante de mitigadores modernos. Wiebing *et al.* (2024), com o InSpectre Gadget, analisam a superfície residual de *gadgets* Spectre em cenários de privilégios cruzados. Esses estudos reforçam que mecanismos como CET e IBT reduzem a superfície de ataque, mas não eliminam a necessidade de defesas complementares e avaliações empíricas da explorabilidade residual.

A Tabela 3.1, a seguir, registra os trabalhos analisados segundo as categorias em que foram agrupados.

Tabela 3.1 Trabalhos relacionados categorizados.

Trabalho	Categoria 1	Categoria 2	Categoria 3	Categoria 4	Categoria 5
2005 – L. Van Put <i>et al.</i>	X				
2011 – Bernat & Miller				X	
2013 – Smithson <i>et al.</i>	X				
2014 – Zhang <i>et al.</i>				X	
2014 – Tymburibá <i>et al.</i>					X
2016 – Meng & Miller				X	
2016 – Meng & Liu			X		
2016 – Wang <i>et al.</i>			X		

Trabalho	Categoria 1	Categoria 2	Categoria 3	Categoria 4	Categoria 5
2017 – Hawkins <i>et al.</i>	X				
2019 – Wenzl <i>et al.</i>				X	
2019 – Rimsa <i>et al.</i>			X		
2019 – Shanbhogue <i>et al.</i>		X			
2020 – Duck <i>et al.</i>	X				
2021 – Meng <i>et al.</i>		X			
2022 – D’Elia <i>et al.</i>			X		
2022 – Schulte <i>et al.</i>			X		
2022 – Xie <i>et al.</i>		X			
2022 – Barberis <i>et al.</i>					X
2023 – Gaidis <i>et al.</i>		X			
2024 – Wiebing <i>et al.</i>					X
2025 – Zhao <i>et al.</i>				X	
2025 – Kim <i>et al.</i>	X				
2026 – Baumann <i>et al.</i>		X			

Observa-se uma concentração expressiva de estudos voltados à reescrita binária, análise de binários e proteção de fluxo de controle. Essa concentração reforça a atualidade do tema e mostra que a adaptação de executáveis a mecanismos modernos de segurança permanece um desafio relevante. Ao mesmo tempo, a presença de trabalhos recentes sobre ataques de execução transiente e limitações de defesas baseadas em hardware evidencia que a proteção do fluxo de controle não pode ser tratada como um problema completamente resolvido.

Além da categorização geral, a Tabela 3.2 evidencia que os trabalhos mais próximos desta dissertação se relacionam com CET/IBT de formas distintas. O FineIBT utiliza o IBT como base para impor verificações mais finas, mas depende de modificações no compilador e, portanto, de acesso ao código-fonte. O SURI e o TVA exploram informações estruturais presentes em binários modernos, incluindo marcações associadas ao CET, para melhorar *reassembly* ou desmontagem estática, mas não têm como objetivo inserir compatibilidade com IBT em programas legados. O Trio securis mostra que políticas no estilo CET/IBT podem ser combinadas com defesas contra execução especulativa, ampliando o escopo de proteção para

além do fluxo arquitetural. O BIN2CET diferencia-se por atuar diretamente sobre binários ELF legados desprotegidos, inserindo *endbr64* por reescrita estática local e preservando instruções sobrescritas por meio de trampolins.

A comparação mostra que o BIN2CET ocupa uma posição distinta na literatura. Diferentemente de abordagens baseadas em compilador, ele não exige código-fonte. Diferentemente de trabalhos voltados principalmente à desmontagem ou a remontagem de binários, ele não se limita a interpretar melhor binários já existentes, mas transforma o executável para torná-lo compatível com os requisitos estruturais do IBT. Essa distinção fundamenta a contribuição desta dissertação: propor e avaliar uma metodologia prática de reescrita binária estática para compatibilizar binários ELF legados com uma defesa de hardware moderna, preservando a funcionalidade observada e reduzindo a superfície efetiva de destinos válidos para transferências indiretas.

Tabela 3.2 Comparação entre trabalhos relacionados a CET/IBT e a proposta desta dissertação.

Ferramenta	Ano	Funcionamento	Necessário Código-fonte?	Relação com CET / IBT
FineIBT	2023	Modifica o compilador LLVM para inserir verificações adicionais de tipo, aridade e contexto em transferências indiretas.	Sim	Usa o IBT como base e adiciona checagens extras por software para restringir destinos válidos.
SURI	2025	Realiza remontagem de binários x86-64 modernos, com foco na simbolização de referências e preservação da semântica.	Não	Usa marcações IBT que já existem no binário como informação estrutural auxiliar.
TVA	2025	Executa desmontagem estática mais precisa explorando propriedades introduzidas pela CET.	Não	Usa <i>endbr64</i> como indicação de destino válido e descarta caminhos que não respeitam essa propriedade.
Triosecuris	2026	Propõe uma defesa formalmente verificada contra sequestro especulativo de fluxo de controle, combinando CFI assistido por hardware e <i>speculative load hardening</i> .	Sim	Usa uma política no estilo CET/IBT como base e combina essa proteção com mitigação contra execução especulativa.

Ferramenta	Ano	Funcionamento	Necessário Código-fonte?	Relação com CET / IBT
BIN2CET	2026	Aplica reescrita binária estática local com microtransformações, trampolins e correção de instruções realocadas.	Não	Insere <i>endbr64</i> em binários ELF legados desprotegidos e usa <i>notrack</i> como <i>fallback</i> conservador quando o alvo não pode ser reescrito com segurança.

A comparação apresentada na Tabela 3.2 também ajuda a justificar a escolha por trampolins no BIN2CET. Trabalhos como SURI e TVA lidam com problemas importantes de reconstrução, remontagem ou desmontagem de binários modernos, mas não têm como objetivo principal transformar binários ELF legados desprotegidos em binários compatíveis com IBT. Em particular, quando marcações CET/IBT já existem no executável, elas podem ser exploradas como informação estrutural auxiliar. O cenário tratado nesta dissertação é diferente: parte-se de binários que não possuem *endbr64* nos destinos indiretos e que não podem ser recompilados. Nesse contexto, inserir uma nova instrução em código já compilado pode sobrescrever bytes de instruções existentes. Os trampolins são, portanto, o mecanismo usado para preservar a semântica dessas instruções deslocadas e permitir que o binário seja modificado sem quebrar o fluxo original de execução.

Dessa forma, o conjunto de trabalhos analisados fornece uma base sólida para a proposta desenvolvida nos capítulos seguintes desta dissertação, que visa aplicar técnicas realistas de reescrita binária para compatibilização de binários legados com tecnologias modernas de defesa como o CET. Os próximos capítulos se dedicarão à apresentação da proposta desenvolvida e dos experimentos que buscam validar sua viabilidade e eficácia em cenários concretos.

4 Sistema BIN2CET: Compatibilização com IBT por Reescrita Binária Estática

Este capítulo apresenta o BIN2CET, um sistema automatizado de reescrita binária estática desenvolvido para compatibilizar binários ELF legados com o mecanismo *Indirect Branch Tracking* (IBT) da Intel CET. A proposta parte do desafio de introduzir marcações estruturais exigidas pelo IBT em executáveis já compilados, sem acesso ao código-fonte, sem recompilação e sem depender da recuperação completa do grafo de fluxo de controle.

Inicialmente, discute-se o problema técnico da reescrita em nível de instrução, destacando as limitações impostas pelo layout fixo do binário e pela necessidade de preservar a semântica original do programa. Em seguida, são descritos o fluxo de análise binária, a geração de artefatos intermediários, as estratégias de transformação adotadas pelo sistema e a etapa de validação dos binários instrumentados. Essa organização permite mostrar como o BIN2CET evolui de transformações locais sobre instruções para um *pipeline* automatizado de compatibilização com IBT.

4.1 Problema de Compatibilização

A compatibilização de binários legados com o mecanismo IBT impõe desafios específicos porque os executáveis já possuem layout de código definido, instruções organizadas em endereços fixos e ausência, total ou parcial, de informações estruturais disponíveis em tempo de compilação. Diferentemente de um cenário de recompilação, no qual o compilador pode inserir instruções *endbr64* nos pontos de entrada apropriados, a reescrita binária precisa introduzir essas marcações diretamente no código de máquina existente, respeitando fronteiras de instrução, permissões de segmento e dependências de endereçamento.

Esse problema se torna mais complexo quando a inserção de uma instrução ou de um redirecionamento sobrescreve bytes pertencentes a instruções originais do programa. Nesses casos, não basta modificar o ponto de entrada: é necessário preservar as instruções deslocadas e garantir que elas continuem produzindo o mesmo efeito semântico. Essa preservação é particularmente crítica para instruções sensíveis à posição, como aquelas que utilizam endereçamento relativo a RIP, pois seu comportamento depende do endereço em que são executadas.

Além disso, a recuperação completa do grafo de fluxo de controle em binários reais é uma tarefa sujeita a imprecisões, especialmente em executáveis otimizados, sem símbolos ou com padrões complexos de transferência indireta. Por essa razão, o BIN2CET adota uma estratégia pragmática: instrumentar os alvos recuperados com segurança, preservar a semântica por meio de trampolins quando necessário e empregar *notrack* como *fallback* conservador nos casos em que a reescrita segura do alvo não pode ser garantida.

4.2 Metodologia Proposta

A metodologia proposta organiza a compatibilização de binários ELF legados com o mecanismo IBT como um fluxo automatizado de análise, transformação e validação. Esse fluxo foi materializado no BIN2CET, um sistema desenvolvido para receber um binário originalmente desprotegido, identificar pontos relevantes para instrumentação, gerar artefatos intermediários de reescrita e produzir um novo binário compatível com os requisitos estruturais do IBT.

A metodologia parte da premissa de que, em binários reais, não é viável depender da recuperação completa do grafo de fluxo de controle. Essa limitação é ainda mais evidente em executáveis otimizados, sem símbolos ou com padrões complexos de chamadas e saltos indiretos. Por essa razão, o BIN2CET privilegia a identificação de estruturas recuperáveis, como entradas de função e locais de transferências indiretas, e adota tratamento conservador nos casos em que a reescrita segura do alvo não pode ser garantida.

A Figura 4.1 apresenta uma visão geral do *pipeline* implementado pelo BIN2CET, destacando as etapas de análise, geração de artefatos, transformação e saída/validação. O fluxo foi organizado em quatro etapas principais, adaptadas do modelo descrito por Wenzl *et al.* (2019): análise preliminar, análise estrutural, transformação e geração do binário instrumentado. Na etapa de análise preliminar, o binário ELF de entrada é examinado para extração de informações estruturais necessárias à instrumentação. Na etapa de análise estrutural, essas informações são organizadas em um artefato intermediário, representado pelo arquivo `analysis.json`, que registra entradas de função, transferências indiretas, tamanhos de instruções, offsets e demais metadados relevantes para a reescrita. Na etapa de transformação, o sistema combina manipulação em nível de ELF, por meio da biblioteca LIEF, com a geração de trampolins e redirecionamentos por meio de uma versão estendida do E9Patch. Por fim, a etapa de saída produz o binário instrumentado e permite verificar se o executável resultante preserva o comportamento esperado.

O processo inicia com um binário ELF original, analisado por mecanismos baseados em objdump ou Ghidra. Em seguida, os registros estruturais extraídos são consolidados no arquivo analysis.json e utilizados pelo módulo de transformação, implementado em retrofit_engine.py. Essa etapa concentra as principais modificações introduzidas pelo sistema, incluindo a inserção de instruções *endbr64*, o uso de *notrack* quando necessário e a correção de referências relativas a RIP em instruções movidas para trampolins. Assim, a compatibilização proposta não consiste apenas em inserir marcadores de destino, mas também em preservar o fluxo de controle e os acessos a dados quando instruções precisam ser deslocadas.

As subseções seguintes descrevem, em maior detalhe, as etapas de análise, geração de artefato, transformação e saída/validação que compõem o *pipeline* do BIN2CET.

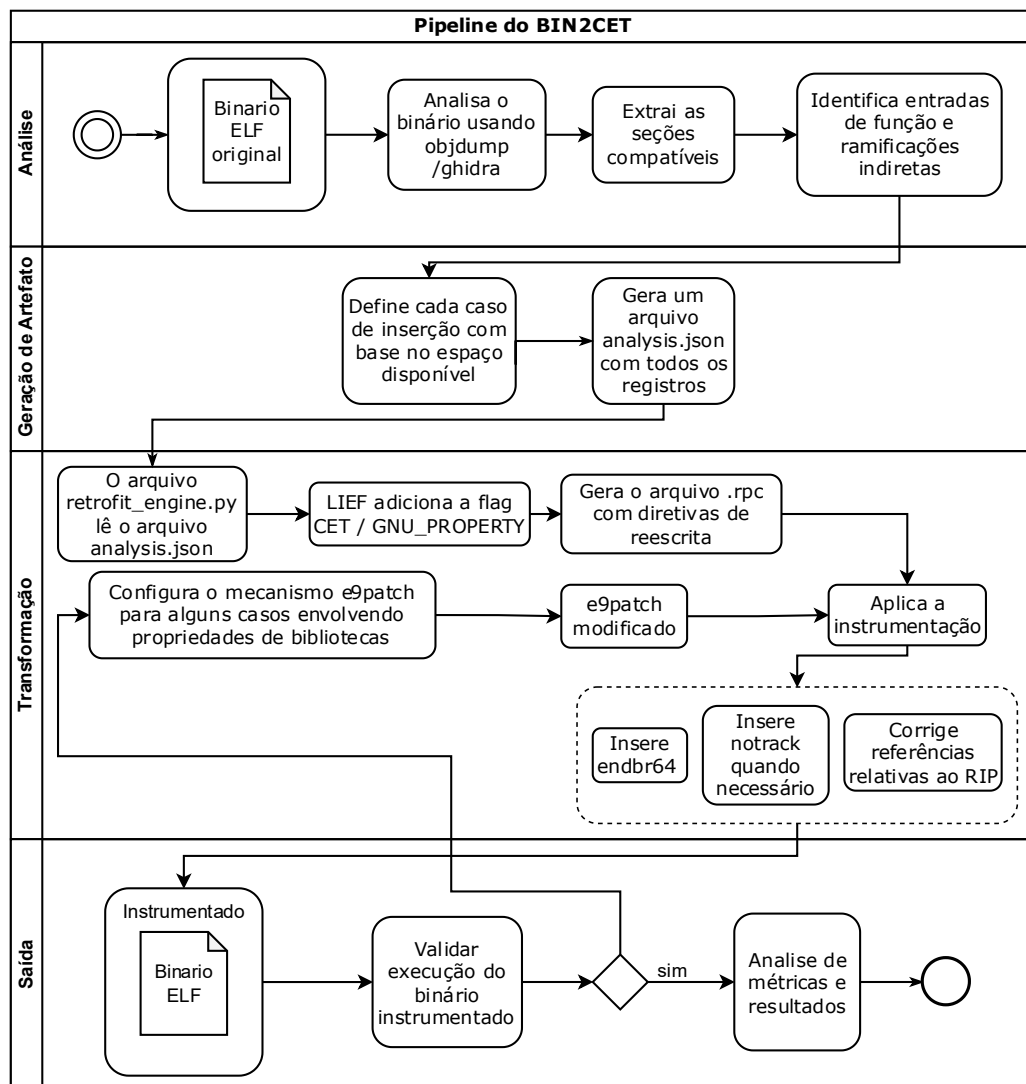


Figura 4.1 Fluxograma da metodologia com as etapas da reescrita binária estática.

4.2.1 Análise

A primeira etapa do fluxo metodológico consiste na análise do binário ELF original, com o objetivo de recuperar as informações estruturais necessárias para orientar a instrumentação. Essa etapa identifica entradas de função, locais de chamadas e saltos indiretos, tamanhos de instruções, offsets em arquivo, seções executáveis e instruções que exigem tratamento especial por dependerem de sua posição original no código. Essas informações são essenciais porque a compatibilização com IBT não pode ser realizada por simples inserção textual de instruções. Em um binário já compilado, cada modificação precisa respeitar fronteiras de instrução, permissões de segmento e a semântica do trecho reescrito.

No BIN2CET, essa etapa é conduzida pelos analisadores baseados em objdump e Ghidra. O analisador baseado em objdump é mais leve e adequado quando a desmontagem textual fornece informações suficientes para identificar entradas de função e instruções de transferência indireta. O analisador baseado em Ghidra, por sua vez, é útil em cenários nos quais a recuperação estrutural exige uma análise mais rica de funções, blocos básicos e contexto das instruções. A escolha entre os analisadores reflete um compromisso entre custo de análise e qualidade da recuperação estrutural, sem alterar o objetivo principal da metodologia.

O propósito dessa análise não é reconstruir perfeitamente o grafo de fluxo de controle do programa. Em binários reais, principalmente quando símbolos são removidos ou quando otimizações de compilador modificam a organização do código, essa reconstrução pode ser imprecisa. Por essa razão, a metodologia busca recuperar informações suficientes para orientar transformações seguras. Entradas de função e locais de transferências indiretas são tratados como candidatos à instrumentação, enquanto situações ambíguas ou inseguras são encaminhadas para tratamento conservador nas etapas posteriores.

Em alguns casos, a análise binária permite relacionar o ponto de instrumentação a construções reconhecíveis em nível de código-fonte, como *callbacks*, ponteiros de função, métodos virtuais ou tabelas de despacho. Por exemplo, uma chamada indireta gerada a partir de um *callback* pode ser mapeada para a entrada da função-alvo, tornando natural a inserção de *endbr64* nesse destino. No entanto, esse mapeamento não é uma garantia geral da abordagem. O BIN2CET não depende da reconstrução completa do código-fonte nem do grafo de fluxo de controle; ele opera sobre evidências estruturais recuperadas do binário, como entradas de função, instruções de chamada ou salto indireto, offsets de arquivo, bytes de instruções e regiões sensíveis à realocação. Assim, quando o vínculo com uma abstração de alto nível é recuperável,

ele auxilia a interpretação e validação da transformação, mas a decisão de instrumentação permanece baseada nas propriedades concretas do binário.

4.2.2 Geração de Artefato Intermediário

Após a análise, as informações recuperadas são organizadas em um artefato intermediário que atua como interface entre os analisadores e o mecanismo de transformação. No *pipeline* do BIN2CET, esse artefato é representado pelo arquivo *analysis.json*, responsável por consolidar os elementos necessários para a reescrita do binário. Entre esses elementos estão os endereços das entradas de função recuperadas, os locais de chamadas e saltos indiretos, o tamanho das instruções, os offsets correspondentes no arquivo ELF e os metadados necessários para decidir qual estratégia de instrumentação deve ser aplicada em cada caso.

Essa etapa é importante porque desacopla a análise da transformação. Em vez de tornar o sistema dependente de uma única fonte de informação estrutural, o BIN2CET passa a operar sobre uma representação intermediária padronizada. Assim, diferentes analisadores podem ser integrados ao *pipeline*, desde que produzam registros compatíveis com o formato esperado. Essa organização também favorece a reprodução dos experimentos, pois o arquivo gerado documenta os pontos identificados, as informações estruturais e as decisões que orientam a etapa de reescrita.

A geração do artefato também permite tratar cada caso de instrumentação de acordo com as condições encontradas no binário. Quando uma entrada de função recuperada possui espaço suficiente e pode ser reescrita com segurança, ela se torna candidata à inserção de *endbr64*. Quando o alvo não pode ser normalizado diretamente, a informação registrada no artefato permite que a etapa de transformação decida pelo uso de um mecanismo conservador, como o uso do *notrack*. Assim, o arquivo intermediário não representa apenas uma listagem de endereços, mas uma base de decisão para a aplicação das estratégias de compatibilização.

4.2.3 Transformação

A etapa de transformação é responsável por aplicar as modificações necessárias para tornar o binário compatível com os requisitos estruturais do IBT. No BIN2CET, essa etapa combina manipulação em nível de ELF com reescrita de código executável. A manipulação do formato ELF é realizada com apoio da biblioteca LIEF, utilizada para alterar metadados, inserir anotações e apoiar substituições pontuais no binário. O LIEF é responsável por adicionar a propriedade GNU associada ao CET e por alterar diretamente a seção *.text*, inserindo *endbr64*

nas entradas selecionadas e preenchendo bytes remanescentes com *nop* quando necessário. Em seguida, o BIN2CET gera um arquivo *.rpc* contendo comandos JSON-RPC para o E9Patch. Esse arquivo descreve o binário de entrada, as instruções selecionadas, os trampolins, as instruções sobrescritas que precisam ser realocadas e o binário final a ser emitido. Para preservar a semântica, instruções movidas para trampolins são associadas ao endereço original, permitindo recalcular operandos dependentes da posição, como referências relativas a RIP. Quando uma transferência indireta é tratada por *fallback*, o E9Patch modificado materializa o prefixo *notrack* na instrução correspondente, tornando-a compatível com o comportamento esperado pelo IBT.

A estratégia preferencial de transformação consiste em inserir a instrução *endbr64* nos alvos recuperados de transferências indiretas. Quando o destino identificado corresponde a uma entrada de função ou a uma região que pode ser reescrita com segurança, o BIN2CET substitui a região inicial por um ponto de entrada válido para a política do IBT. A instrução *endbr64* ocupa 4 *bytes*; quando a preservação das instruções originais exige um redirecionamento adicional, o patch pode demandar uma região maior, tipicamente associada à combinação entre marcador de entrada, preenchimento com instruções *nop* e salto para trampoline.

Caso a inserção sobrescreva instruções originais, essas instruções são deslocadas para um trampoline, executadas nessa região auxiliar e, em seguida, o controle retorna ao ponto adequado do fluxo original. Regiões como PLT, código gerado dinamicamente, funções sem símbolos e layouts altamente otimizados exigem tratamento cauteloso, pois podem limitar a recuperação estrutural e o posicionamento seguro de trampolins. Esse procedimento permite tornar o destino compatível com o IBT sem descartar o comportamento previamente implementado naquele trecho de código.

O uso de trampolins é necessário porque o BIN2CET atua sobre executáveis já compilados, nos quais o layout do código está previamente definido. Diferentemente de um cenário de recompilação, em que o compilador pode reorganizar o código e inserir *endbr64* nos pontos apropriados, a reescrita binária precisa operar sobre bytes já alocados no executável. Assim, a inserção de uma nova instrução ou de um salto de redirecionamento pode sobrescrever bytes pertencentes a instruções originais. O trampoline permite preservar essas instruções em uma região auxiliar e restabelecer o fluxo de execução após sua execução. Portanto, o custo estrutural introduzido pelos trampolins decorre da necessidade de modificar binários legados sem acesso ao código-fonte e sem reconstrução completa do grafo de fluxo de controle.

Um dos principais desafios dessa etapa ocorre quando as instruções deslocadas dependem de sua posição original em memória. Esse é o caso de instruções com endereçamento

relativo a RIP, nas quais o endereço efetivo acessado é calculado a partir da posição da própria instrução no código. Se uma instrução desse tipo fosse simplesmente copiada para um trampolim, o deslocamento passaria a ser interpretado em relação ao novo endereço, podendo causar acesso a uma variável global, literal ou estrutura de dados diferente da originalmente pretendida. Para evitar esse problema, o suporte de realocação do BIN2CET recalcula os deslocamentos necessários antes da emissão do binário final, de modo que a instrução realocada preserve o mesmo alvo semântico observado no programa original.

Nem todos os alvos podem ser reescritos com segurança. Em binários otimizados, a região inicial de uma função pode ser compacta demais para acomodar a instrumentação sem violar fronteiras de instrução, ou o E9Patch pode não encontrar uma posição adequada para o trampolim. Nessas situações, o BIN2CET utiliza *notrack* como mecanismo conservador de compatibilidade. Esse uso deve ser entendido como *fallback*, e não como estratégia preferencial, pois *notrack* desabilita a checagem de destino para uma transferência específica. Embora preserve a execução correta do binário, essa alternativa é menos restritiva do ponto de vista de segurança do que a marcação explícita do alvo com *endbr64*. Por essa razão, a metodologia prioriza a normalização dos destinos e reserva *notrack* para os casos em que a reescrita segura do alvo não é viável.

4.2.4 Saída e Validação

A etapa final do *pipeline* consiste na geração do binário instrumentado e na verificação de seu comportamento. Ao término da transformação, o BIN2CET produz um novo binário ELF contendo as modificações necessárias à compatibilização com o IBT. Esse binário incorpora os pontos de entrada marcados com instruções *endbr64*, os redirecionamentos para trampolins, as instruções realocadas com seus deslocamentos ajustados e, quando necessário, o uso de *notrack* em transferências indiretas tratadas de forma conservadora.

A validação do binário resultante ocorre em duas frentes complementares. A primeira verifica se o executável continua funcionando conforme esperado, preservando a saída e o comportamento observável do programa original. A segunda verifica se as transferências indiretas observadas durante a execução alcançam destinos compatíveis com a política adotada, isto é, se chegam a pontos iniciados por *endbr64* ou se foram explicitamente tratadas com *notrack*. Nos experimentos desta dissertação, essa validação é conduzida por meio da execução dos programas instrumentados, da suíte de testes do Coreutils e de um verificador em tempo de execução baseado em PIN.

Além de produzir o binário final, essa etapa gera os dados necessários para a avaliação experimental. A partir dos binários instrumentados e dos resultados de validação, torna-se possível medir a cobertura de instrumentação, o *overhead* de execução, o crescimento do tamanho binário e a redução da superfície efetiva de ataque. Dessa forma, a saída do *pipeline* não se limita ao artefato executável, mas também fornece a base empírica usada para avaliar a viabilidade e os limites da metodologia proposta.

4.3 Considerações Finais

Este capítulo apresentou o BIN2CET, um sistema automatizado de reescrita binária estática voltado à compatibilização de binários ELF legados com o mecanismo IBT da Intel CET. A proposta foi organizada como um *pipeline* composto pelas etapas de análise, geração de artefato intermediário, transformação e saída/validação, permitindo estruturar o processo de compatibilização sem exigir acesso ao código-fonte nem recuperação completa do grafo de fluxo de controle.

Ao longo do capítulo, foram discutidas as principais decisões técnicas adotadas pelo sistema, incluindo a identificação de entradas de função e transferências indiretas, a geração do arquivo `analysis.json`, a inserção de instruções *endbr64*, o uso de trampolins para preservação de instruções sobrescritas, o tratamento de instruções sensíveis à posição e a aplicação conservadora de *notrack* quando a reescrita segura do alvo não é possível. Essas estratégias permitem adaptar binários já compilados aos requisitos estruturais do IBT, preservando o comportamento observável do executável sempre que as condições de reescrita são satisfeitas.

A metodologia proposta evidencia que a compatibilização com IBT não se resume à inserção de marcadores de destino, pois exige também a preservação do fluxo de controle, dos acessos a dados e da semântica das instruções deslocadas. Dessa forma, o BIN2CET fornece a base técnica necessária para investigar, de forma experimental, a eficácia da instrumentação, seus custos de desempenho e tamanho binário, bem como seu impacto na redução da superfície efetiva de ataque. Esses aspectos são avaliados no capítulo seguinte.

5 Experimentos e Resultados

Este capítulo apresenta a avaliação experimental do BIN2CET, sistema proposto nesta dissertação para compatibilizar binários ELF legados com o mecanismo IBT da Intel CET por meio de reescrita binária estática. A avaliação foi estruturada para verificar a capacidade do sistema de instrumentar pontos relevantes do fluxo de controle indireto, preservar o comportamento funcional dos programas transformados, mensurar os custos de desempenho e de tamanho binário, e analisar o impacto da técnica sobre a superfície efetiva de ataque disponível para exploração por reutilização de código.

A experimentação foi orientada pelas questões de pesquisa definidas no Capítulo 1 e conduzida sobre um conjunto de programas reais da suíte GNU Coreutils. Ao todo, foram utilizados 102 programas, compilados como binários PIE (*Position-Independent Executable*) e avaliados sob diferentes níveis de otimização. Essa escolha permite observar o comportamento da abordagem em executáveis com estruturas distintas de fluxo de controle, diferentes tamanhos, diferentes padrões de uso de funções e diferentes layouts produzidos pelo compilador.

5.1 Protocolo Experimental

O protocolo experimental foi estruturado para reproduzir um cenário em que binários legados precisam ser adaptados para compatibilidade com IBT sem acesso ao código-fonte original durante a etapa de transformação. Para isso, os programas foram inicialmente compilados sem proteção CET, analisados para extração de informações estruturais, transformados pelo BIN2CET e, posteriormente, avaliados quanto à compatibilidade, desempenho, tamanho e superfície de ataque.

O fluxo experimental foi dividido em quatro etapas principais. Na primeira etapa, todos os programas da suíte Coreutils foram compilados sem proteção CET, utilizando a opção de compilação `-fcf-protection=none`. Os binários resultantes foram gerados como *Position Independent Executable* (PIE). Embora o BIN2CET possa ser aplicado a binários não PIE, a avaliação concentrou-se em PIE porque esse formato é amplamente utilizado em distribuições modernas e porque o E9Patch apresenta melhor efetividade nesse tipo de executável.

Na segunda etapa, os binários desprotegidos foram analisados para extração de entradas de função, chamadas indiretas e saltos indiretos. Essa análise foi conduzida principalmente por meio do analisador baseado em `objdump`, por ser mais leve e adequado ao processamento em

escala. A comparação com o analisador baseado em Ghidra indicou resultados semelhantes no conjunto avaliado, o que justificou a adoção do analisador mais rápido durante a execução completa dos experimentos.

Na terceira etapa, as informações extraídas foram fornecidas ao BIN2CET para aplicação das transformações necessárias. O sistema tentou inserir *endbr64* nos alvos recuperados, preservar instruções sobrescritas por meio de trampolins e corrigir referências sensíveis à realocação quando necessário. Nos casos em que uma entrada de função ou destino indireto não pôde ser reescrito com segurança, o próprio desvio indireto pôde ser tratado com *notrack*. Essa estratégia permitiu combinar instrumentação preferencial no destino com tratamento conservador quando a normalização completa do alvo não era viável.

Na quarta etapa, os binários instrumentados foram executados para verificar a preservação funcional e a conformidade com os requisitos observados de IBT. Para isso, utilizou-se a suíte de testes do Coreutils e um verificador em tempo de execução baseado em PIN, capaz de monitorar chamadas e saltos indiretos durante a execução. Esse verificador observa se os destinos alcançados começam com *endbr64* ou se a transferência foi explicitamente tratada com *notrack*. Dessa forma, a instrumentação dinâmica não é usada como mecanismo principal de proteção ou compatibilização, mas como recurso de validação experimental. Sua função é agregar evidência em tempo de execução sobre o comportamento efetivo dos desvios indiretos, permitindo responder à Q2 no que se refere à integridade do fluxo de controle após a aplicação das microtransformações estáticas.

Alguns testes do Coreutils e quatro programas (*timeout*, *env*, *nice* e *stty*) foram removidos dessa etapa específica da avaliação devido a incompatibilidades com PIN. Essa remoção não significa que tais programas tenham sido excluídos do corpus de transformação, mas apenas que não foi possível utilizá-los de forma confiável na validação dinâmica. Por fim, foram coletadas métricas de desempenho com o Linux *perf* e métricas de superfície de ataque com a ferramenta ROPgadget.

5.1.1 Ambiente e Ferramentas Utilizadas

A avaliação foi conduzida em um ambiente baseado em contêiner Docker, com o objetivo de tornar o fluxo experimental reprodutível e reduzir interferências decorrentes de diferenças de configuração entre execuções. O ambiente reúne as ferramentas necessárias para compilar os programas, executar os analisadores, aplicar a reescrita binária, validar os binários instrumentados e extrair as métricas utilizadas na avaliação.

O BIN2CET utiliza a biblioteca LIEF para manipulação em nível de ELF, incluindo ajustes estruturais no binário e inserção de propriedades associadas à compatibilidade com IBT. Para a reescrita do código executável, o sistema utiliza uma versão estendida do E9Patch, responsável por materializar trampolins, redirecionar o fluxo de controle e preservar instruções sobrescritas. A análise dos binários é realizada por scripts baseados em *objdump* e, quando necessário, Ghidra, com o objetivo de identificar entradas de função, chamadas e saltos indiretos, tamanhos de instruções e demais informações necessárias para orientar a transformação. A validação em tempo de execução é realizada por meio de um verificador baseado em PIN, usado para observar o comportamento efetivo das transferências indiretas durante a execução dos programas. Esse verificador complementa a análise estática ao registrar os destinos realmente alcançados pelas cargas de teste, permitindo verificar se eles correspondem a pontos iniciados por *endbr64* ou a transferências explicitamente tratadas com *notrack*. Assim, o PIN atua como instrumento de observação e validação, e não como parte da transformação aplicada pelo BIN2CET.

Para a coleta de métricas de desempenho, utiliza-se o Linux *perf*, com foco em contadores como ciclos, task-clock, ciclos parados no front-end, desvios, falhas de predição de desvio, referências de cache e faltas de cache. Para a análise da superfície de ataque, utiliza-se a ferramenta ROPgadget, configurado para realizar uma enumeração sintática ampla de *gadgets*.

5.1.2 Procedimento de Medição e Tratamento Estatístico

As medições de desempenho foram conduzidas separadamente para cada nível de otimização e para cada variante do binário avaliado. Foram considerados os níveis -O0, -O2 e -O3, sempre comparando a versão original do programa com a versão instrumentada pelo BIN2CET. Para cada programa, variante e nível de otimização, foram realizadas 15 repetições, permitindo observar tanto o valor médio dos contadores coletados quanto a dispersão das medições.

A coleta foi realizada por meio de *wrappers* compilados automaticamente para cada programa. Durante a execução da suíte de testes do Coreutils com *make check*, cada chamada a um programa era redirecionada para um wrapper, que preservava os argumentos originais do teste e executava o binário real sob *perf stat*. Dessa forma, a unidade básica de medição corresponde a uma invocação concreta de um programa durante a suíte de testes, e não apenas ao binário considerado de forma agregada.

Dos 102 programas do Coreutils, os programas *timeout*, *env*, *nice* e *stty* foram excluídos das medições dinâmicas por restrições do mecanismo de execução usado nos *wrappers* de análise, resultando em 98 programas considerados nas medições com perf. A saída de cada execução foi armazenada em um log individual, posteriormente consolidado por programa, variante e nível de otimização.

Os eventos de desempenho foram divididos em dois grupos de coleta. O grupo perf1 reuniu os eventos instructions, cycles, task-clock e stalled-cycles-frontend, enquanto o grupo perf2 reuniu branch-misses, branches, cache-misses e cache-references. Essa separação foi adotada devido ao número limitado de contadores físicos de desempenho disponíveis no processador. Quando muitos eventos de hardware são solicitados simultaneamente, o perf pode recorrer à multiplexação, alternando a coleta entre eventos e produzindo valores estimados. Ao dividir os eventos em dois grupos, buscou-se reduzir esse efeito e preservar maior precisão nas medições.

A partir dos logs individuais, os resultados foram consolidados por programa, variante e nível de otimização. Para cada métrica, o overhead foi calculado comparando a média das 15 repetições da versão instrumentada com a média das 15 repetições da versão original. O overhead percentual foi calculado pela razão entre a diferença observada e o valor médio da versão original. Valores positivos indicam aumento do custo após a instrumentação, enquanto valores negativos indicam redução do valor medido em relação ao binário original.

Além das médias, foram calculadas medidas de dispersão para caracterizar a variabilidade dos resultados. No nível de cada programa, as 15 repetições permitiram observar a estabilidade das medições individuais. No nível agregado, os overheads por programa foram utilizados para calcular média, desvio padrão e intervalo de confiança de 95%, permitindo avaliar se os custos observados representam uma tendência consistente no conjunto avaliado ou se decorrem de poucos casos extremos. Os resultados agregados são discutidos no corpo deste capítulo, enquanto os dados detalhados por programa são apresentados no Apêndice A.

5.1.3 Métricas

As métricas adotadas foram definidas para avaliar a abordagem sob três dimensões complementares: efetividade da compatibilização com o IBT, custo da transformação e impacto sobre a superfície de ataque.

A primeira dimensão mede a efetividade da compatibilização com IBT. Para isso, foram considerados o número de entradas de função recuperadas, o número de alvos protegidos com

instruções *endbr64*, o número de transferências indiretas tratadas com *notrack* e a ocorrência de violações observadas em tempo de execução pelo verificador baseado em PIN. Essa métrica permite avaliar se os binários instrumentados passam a atender os requisitos estruturais do IBT nos comportamentos observados durante os testes.

A segunda dimensão avalia o custo da reescrita. Para o desempenho, foram analisados os contadores coletados com a ferramenta *perf*. Para o impacto estrutural, foi medido o crescimento do tamanho binário antes e depois da instrumentação. Essas métricas permitem avaliar o custo imposto pela inserção de *endbr64*, redirecionamentos, trampolins e instruções realocadas.

A terceira dimensão analisa a superfície efetiva de ataque. Para isso, o ROPgadget foi executado com uma configuração agressiva, usando *--depth 100* e *--multibr*, sem filtros adicionais. Essa configuração favorece uma enumeração ampla de sequências sintaticamente semelhantes a *gadgets*. No entanto, a interpretação da superfície de ataque considera a distinção entre *gadgets* sintáticos e destinos efetivamente válidos sob a política de IBT. Assim, o resultado relevante não é apenas o número bruto de *gadgets* reportados, mas a diferença entre esse espaço sintático e o conjunto menor de destinos que permanecem alcançáveis por transferências indiretas após a instrumentação.

5.2 Resultados

Os resultados são apresentados em análises complementares. Inicialmente, avalia-se a efetividade do BIN2CET na compatibilização dos binários com os requisitos estruturais do IBT. Em seguida, discute-se a relação entre a estrutura dos programas e a quantidade de pontos que precisam ser protegidos. Na sequência, analisam-se os *overheads* de execução e o impacto no tamanho dos binários. Por fim, avalia-se a redução da superfície efetiva de ataque a partir da distinção entre *gadgets* sintáticos e destinos válidos pela política de instrumentação.

5.2.1 Efetividade da Compatibilização com IBT

A primeira análise avalia a efetividade da compatibilização com IBT. Conforme descrito no protocolo experimental, os binários foram analisados por um mecanismo baseado em *objdump*, e as informações extraídas foram fornecidas ao BIN2CET para aplicação das transformações. A avaliação concentrou-se na instrumentação de entradas de função recuperadas e no tratamento de transferências indiretas, considerando que grande parte das chamadas indiretas observadas em programas compilados convencionalmente tem como

destino entradas de função. Dessa forma, a estratégia prioriza a normalização dos destinos com instruções *endbr64* e evita o uso desnecessário de *notrack* quando o alvo pode ser reescrito com segurança.

No nível de otimização -O0, o BIN2CET foi capaz de proteger 100% das entradas de função recuperadas, totalizando quase 20 mil pontos instrumentados no conjunto avaliado. Esse resultado está associado à presença mais frequente de prólogos clássicos de função, que oferecem espaço suficiente para acomodar a instrução *endbr64* e o redirecionamento necessário à preservação das instruções sobrescritas. Em binários otimizados, esse comportamento se altera porque o compilador tende a produzir entradas de função mais compactas, o que reduz a disponibilidade de bytes consecutivos para a reescrita segura.

A Figura 5.1 apresenta a distribuição percentual de funções protegidas e desprotegidas no nível de otimização -O2. Observa-se que a maior parte das funções permaneceu tratável, com aproximadamente 84% das entradas recuperadas protegidas pelo BIN2CET. A fração desprotegida se concentra em torno de 16% está associada principalmente a entradas de função nas quais a região inicial não oferecia espaço suficiente para acomodar a instrumentação sem violar fronteiras de instrução ou nas quais não foi possível posicionar o trampolim de forma adequada.

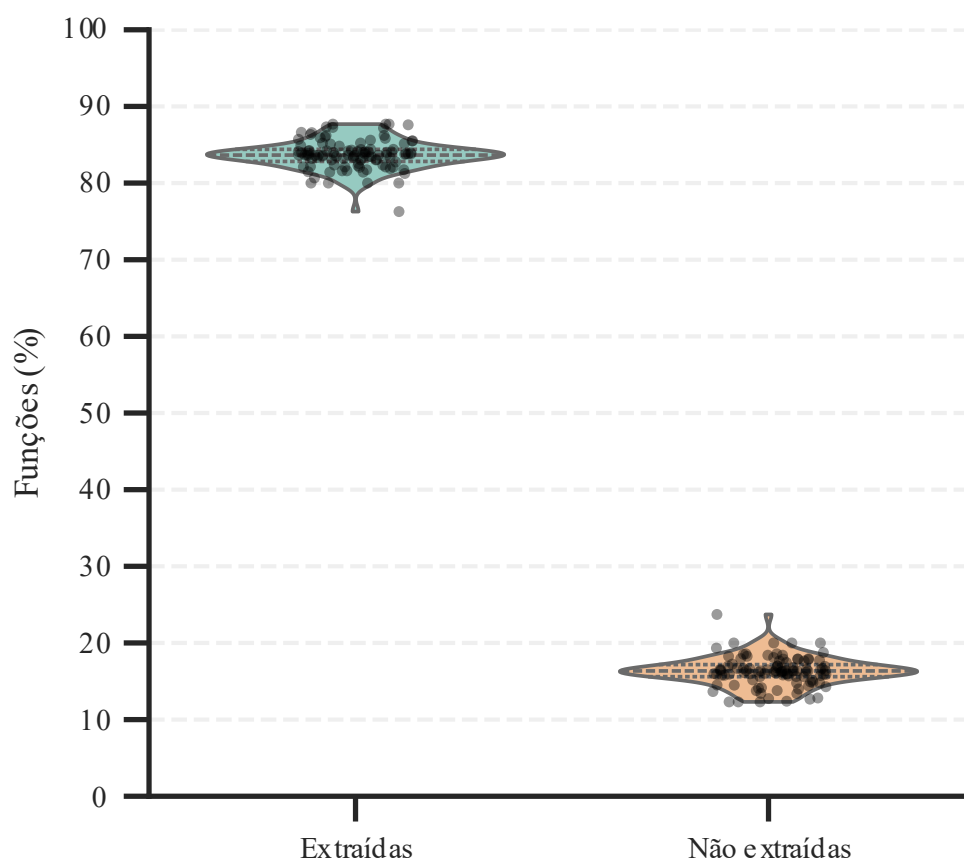


Figura 5.1 Proporção, em %, de funções protegidas e desprotegidas no nível de otimização -O2 para todos os 102 programas do Coreutils.

A Figura 5.2 mostra o mesmo tipo de distribuição para o nível -O3. Nesse cenário, o percentual de funções protegidas permanece elevado, em torno de 82%, embora ligeiramente inferior ao observado em -O2. Esse resultado é coerente com o efeito esperado de otimizações mais agressivas, que tendem a compactar o layout das funções e reduzir oportunidades diretas de reescrita na entrada. Ainda assim, a maior parte das funções recuperadas permaneceu compatível com a estratégia de instrumentação proposta.

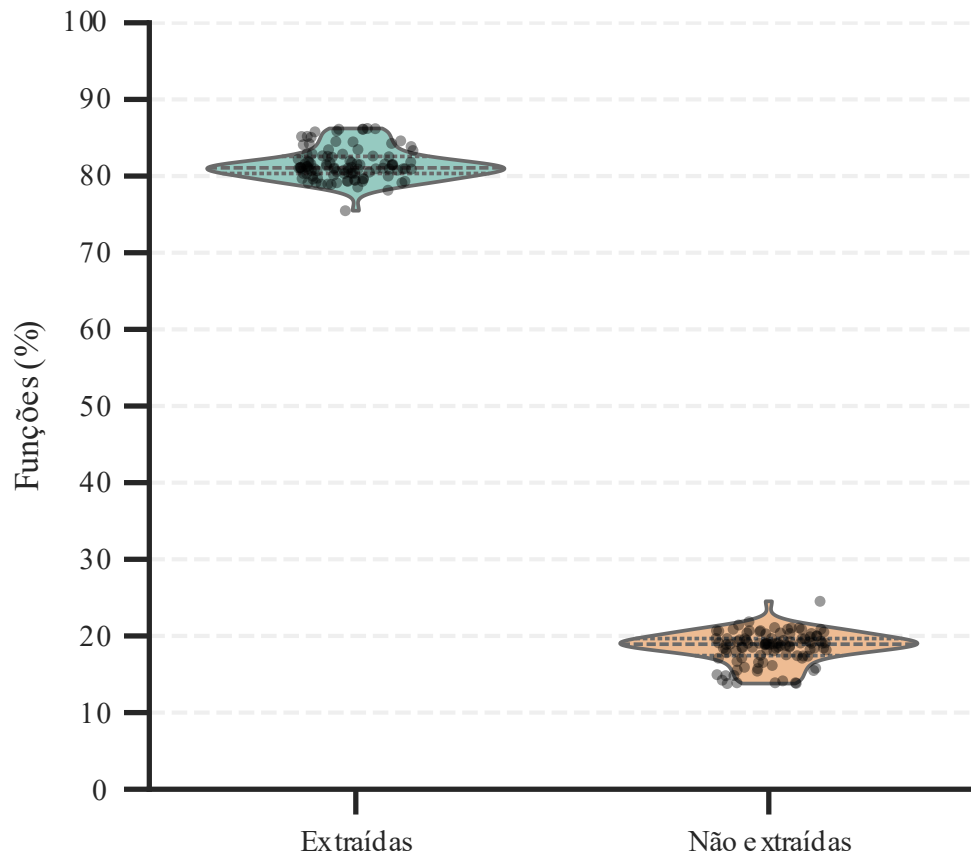


Figura 5.2 Proporção, em %, de funções protegidas e desprotegidas no nível de otimização -O3 para todos os 102 programas do Coreutils.

Após a instrumentação, o verificador de tempo de execução (*runtime checker*) foi usado para identificar se desvios indiretos ainda alcançavam locais sem *endbr64*. Quando esse comportamento era identificado, os desvios correspondentes podiam ser tratados com *notrack*. O uso de *notrack* nesses casos não representa uma falha de tratamento, mas uma decisão conservadora de compatibilidade. A estratégia preferencial do BIN2CET é marcar explicitamente o destino com *endbr64*, preservando a checagem de destino imposta pelo IBT. No entanto, quando a região do alvo não pode ser reescrita com segurança, por falta de espaço, restrições de layout ou impossibilidade de posicionar um trampolim adequado, o uso de *notrack* permite preservar a execução correta do programa. Essa escolha reduz a força da checagem naquele ponto específico, mas evita deixar o binário legado completamente incompatível com IBT. Assim, *notrack* é tratado como *fallback* controlado, aplicado apenas quando a normalização segura do alvo não é viável.

No nível -O0, nenhuma violação de IBT foi observada em tempo de execução. Nos níveis -O2 e -O3, foram identificadas violações nos programas *false* e *true*, cujas funções *main* não puderam ser reescritas. Esse problema ocorre porque a *libc* transfere controle para a função *main* por meio de uma chamada indireta sem uso de *notrack*. Embora a própria *libc* pudesse ser instrumentada, essa possibilidade ficou fora do escopo da avaliação. Um caso semelhante ocorreu apenas em -O2 nos programas *sha224sum*, *sha256sum*, *sha384sum* e *sha512sum*, nos quais um salto indireto não pôde ser instrumentado porque não havia endereço adequado para posicionar o trampolim. Nos casos em que a transformação foi concluída com sucesso, os binários reescritos permaneceram estruturalmente válidos como executáveis ELF e puderam ser carregados e executados no ambiente de testes. As falhas observadas decorreram de limitações específicas de instrumentação, validação dinâmica ou cobertura de bibliotecas externas, e não de uma rejeição generalizada do formato binário produzido. Todos os demais programas avaliáveis com PIN não apresentaram violações observadas em tempo de execução.

Esses resultados indicam que o BIN2CET alcança cobertura completa das entradas de função recuperadas no cenário não otimizado e alta cobertura em layouts otimizados. As limitações observadas estiveram associadas principalmente a restrições práticas da reescrita segura em regiões compactas do código ou à impossibilidade de posicionar trampolins em alguns casos. Ainda assim, a abordagem foi capaz de instrumentar os 102 programas do Coreutils e preservar a execução observada nos testes conduzidos, respeitadas as limitações da validação dinâmica descritas no protocolo experimental.

5.2.2 Relação entre Estrutura do Programa e Pontos de Instrumentação

Além da proporção de funções protegidas, também foi analisada a relação entre a quantidade de entradas de função recuperadas e a quantidade de desvios indiretos identificados nos binários. Essa análise ajuda a compreender se programas com maior estrutura funcional tendem a demandar maior quantidade de pontos relacionados ao tratamento do fluxo de controle indireto.

A Figura 5.3 apresenta a correlação entre o número de entradas de função recuperadas e o número de desvios indiretos nos programas avaliados no cenário -O0. O coeficiente de determinação $R^2 = 0,698$ indica uma correlação positiva relevante entre essas duas variáveis. Em termos práticos, isso sugere que binários com maior quantidade de entradas de função tendem também a apresentar maior número de transferências indiretas que precisam ser

analisadas ou protegidas. Embora a relação não seja perfeita, ela reforça a expectativa de que o custo de instrumentação cresce de forma proporcional à complexidade estrutural do programa.

A dispersão observada também revela que programas com tamanho e estrutura semelhantes podem apresentar quantidades diferentes de desvios indiretos, dependendo de decisões de compilação e do estilo de implementação. O crescimento linear limitado indica que o mecanismo de *fallback* conservador escala de forma previsível com a complexidade binária, garantindo a estabilidade do sistema sem comprometer a camada primária de proteção imposta pelo hardware. Por esse motivo, a metodologia não pode se limitar a métricas globais de tamanho ou número de funções. É necessário identificar diretamente os locais de transferência indireta e decidir, para cada caso, se o destino pode ser normalizado com *endbr64* ou se deve ser tratado de forma conservadora.

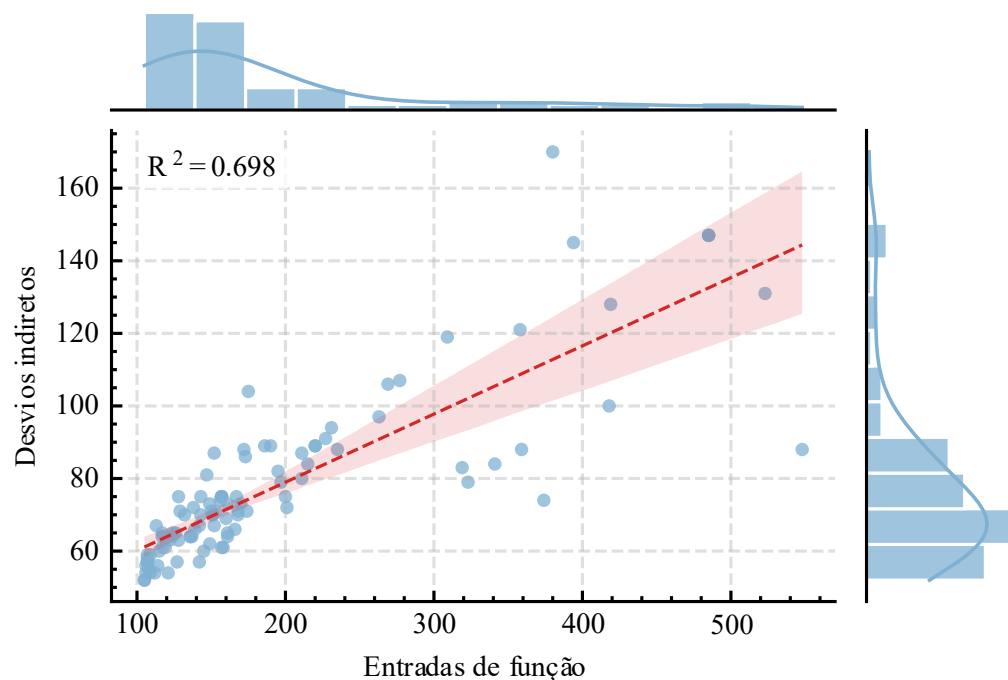


Figura 5.3 Correlação entre as entradas de função recuperadas e os desvios indiretos identificados no cenário -O0.

5.2.3 Overhead de Execução

Para avaliar o custo introduzido pela instrumentação, foram coletados contadores de desempenho com o Linux *perf* durante a execução dos programas da suíte de testes do Coreutils, conforme descrito no protocolo experimental. As medições consideraram os programas avaliáveis por meio dos *wrappers* de execução, excluindo os casos removidos por

incompatibilidades com o mecanismo de validação dinâmica. A análise compara, para cada nível de otimização, a versão original do programa com a versão instrumentada pelo BIN2CET.

A Tabela 5.1 apresenta o *overhead* médio ponderado dos principais contadores, utilizando como peso o número de instruções executadas por programa. Essa agregação permite observar o impacto global da instrumentação considerando a contribuição relativa de cada programa para o volume total de execução. Já a Tabela 5.2 apresenta os overheads médios por programa, agrupados por nível de otimização, incluindo média, desvio padrão, intervalo de confiança de 95%, mínimo e máximo. Nesse caso, cada programa contribui igualmente para a média agregada, o que permite observar a dispersão dos custos entre os binários avaliados.

Tabela 5.1 Overhead médio ponderado, baseado no número de instruções executadas por programa, para contadores do perf.

Eventos do Perf	Otimização -O0			Otimização -O2			Otimização -O3		
	Desprotegido	Protegido	Overhead	Desprotegido	Protegido	Overhead	Desprotegido	Protegido	Overhead
<i>cycles</i>	2265369.63	2782301.94	22.82%	2123523.84	3009095.28	41.70%	2128542.73	3015268.40	41.66%
<i>task-clock</i>	0.81	0.99	22.22%	0.76	1.07	40.79%	0.76	1.07	40.79%
<i>stalled-cycles-frontend</i>	1630736.98	1961997.33	20.31%	1526466.04	2067297.74	35.43%	1530630.42	2071754.38	35.35%
<i>branch-misses</i>	10464.15	13315.61	27.25%	10189.18	14660.20	43.88%	10224.57	14707.28	43.84%
<i>branches</i>	244650.31	307009.47	25.49%	229663.56	349808.99	52.31%	230221.44	349835.22	51.96%
<i>cache-misses</i>	4881.27	5190.76	6.34%	4383.11	4940.14	12.71%	4389.53	4937.34	12.48%
<i>cache-references</i>	29060.60	33540.06	15.41%	27512.60	32725.60	18.95%	27516.50	32733.89	18.96%

Tabela 5.2 Overhead médio de desempenho por nível de otimização, com desvio padrão, intervalo de confiança de 95% e valores mínimo e máximo entre programas.

Otimização	Métrica	N Programas	Média Overhead (%)	Desvio Padrão (p.p)	IC95 % Inferior (%)	IC95 % Superior (%)	Min (%)	Max (%)	Média Ponderada por Valor Original (%)
-O0	<i>instructions</i>	98	33.94	11.79	31.58	36.3	10.6	65.34	30.65
-O0	<i>cycles</i>	98	29.28	8.83	27.51	31.04	9.37	50.73	26.74
-O0	<i>task-clock</i>	98	28.52	9.24	26.67	30.37	7.36	50.85	26.09
-O0	<i>stalled-cycles-frontend</i>	98	26.39	7.52	24.89	27.9	8.33	43.45	24.14
-O0	<i>branches</i>	98	32.61	11.32	30.34	34.88	10.09	63.31	29.52
-O0	<i>branch-misses</i>	98	30.45	7.79	28.89	32.01	14.78	58.93	29.8
-O2	<i>instructions</i>	98	62.41	31.23	56.15	68.67	15.92	194.9	57.92
-O2	<i>cycles</i>	98	48.01	21.44	43.72	52.31	11.67	140.72	44.85
-O2	<i>task-clock</i>	98	46.5	21.22	42.24	50.75	11.87	134.48	43.52
-O2	<i>stalled-cycles-frontend</i>	98	41.23	17.44	37.74	44.73	9.26	116.41	38.52
-O2	<i>branches</i>	98	60.11	30.49	54.0	66.22	15.05	188.05	55.65
-O2	<i>branch-misses</i>	98	46.28	17.73	42.72	49.83	19.84	121.77	45.58
-O3	<i>instructions</i>	98	62.17	29.71	56.21	68.12	15.86	177.17	57.81
-O3	<i>cycles</i>	98	47.83	20.22	43.78	51.89	13.38	126.41	44.8
-O3	<i>task-clock</i>	98	46.9	19.85	42.92	50.88	9.55	122.41	44.07
-O3	<i>stalled-cycles-frontend</i>	98	41.02	16.32	37.75	44.29	11.64	103.92	38.44
-O3	<i>branches</i>	98	59.66	29.23	53.8	65.52	15.0	171.14	55.29
-O3	<i>branch-misses</i>	98	46.51	17.41	43.03	50.0	-2.65	102.87	45.69

Os resultados da Tabela 5.1 indicam aumento consistente nos principais contadores após a instrumentação. Nos contadores associados a tempo de execução, observa-se overhead ponderado de aproximadamente 22% em -O0 e em torno de 41% nos níveis -O2 e -O3, considerando *cycles* e *task-clock*. Esse aumento é acompanhado por crescimento nos eventos associados ao novo fluxo de controle introduzido pela instrumentação, especialmente *branches* e *branch-misses*. Nos níveis -O2 e -O3, o número de desvios executados cresce mais de 50%, enquanto as falhas de predição de desvio aumentam em torno de 44%.

Esse comportamento é compatível com a estratégia de reescrita adotada pelo BIN2CET. A inserção de *endbr64*, redirecionamentos e trampolins aumenta a quantidade de instruções executadas e introduz novas transferências de controle entre o ponto instrumentado, a região auxiliar e o retorno ao fluxo original. Do ponto de vista microarquitetural, esse aumento de desvios pode pressionar estruturas de predição, como o *Branch Target Buffer* (BTB), e elevar o custo associado a falhas de predição. Quando uma predição falha, o processador precisa descartar parte do trabalho especulativo e retomar a execução a partir do destino correto, o que contribui para o aumento de *cycles* e *task-clock*.

A Tabela 5.2 complementa essa análise ao mostrar que o overhead não é uniforme entre os programas avaliados. Em -O0, a média por programa para *cycles* foi de 29,28%, com intervalo de confiança de 95% entre 27,51% e 31,04%. Nos níveis otimizados, os valores médios foram superiores: 48,01% em -O2 e 47,83% em -O3. A dispersão observada, expressa pelo desvio padrão e pelos valores mínimo e máximo, indica que alguns programas são mais sensíveis à instrumentação do que outros. Essa variação é esperada, pois o impacto depende da estrutura de cada binário, da quantidade de pontos instrumentados, do número de trampolins inseridos e do efeito das otimizações do compilador sobre o layout do código.

A diferença entre a Tabela 5.1 e a Tabela 5.2 decorre da estratégia de agregação. Enquanto a Tabela 5.1 apresenta uma média ponderada pelo volume de instruções executadas, a Tabela 5.2 resume os overheads por programa, atribuindo o mesmo peso a cada binário. Assim, a primeira tabela enfatiza o custo agregado observado no conjunto de execuções, enquanto a segunda evidencia a variabilidade entre programas. Em conjunto, as duas análises mostram que a instrumentação impõe um custo mensurável, mas coerente com a natureza da transformação aplicada.

Além dos contadores de tempo, observou-se aumento em eventos associados a desvios, ciclos parados no *front-end* e referências de cache. Esses efeitos são compatíveis com a introdução de caminhos auxiliares de execução e com o aumento do número de instruções

executadas. Portanto, o *overhead* observado deve ser interpretado como parte do custo estrutural da compatibilização de binários já compilados com IBT, especialmente em programas pequenos e de curta duração, como muitos utilitários do Coreutils.

Os resultados detalhados por programa são apresentados no Apêndice A, permitindo identificar os binários que concentraram os maiores e menores overheads em cada nível de otimização

5.2.4 Tamanho do Binário

A análise de tamanho binário complementa a avaliação de desempenho ao mensurar o custo estrutural da reescrita. Como o BIN2CET adiciona marcadores *endbr64*, redirecionamentos, trampolins e instruções realocadas, espera-se que o binário instrumentado cresça em relação à versão original. Esse crescimento, entretanto, deve ser interpretado como parte do custo de preservar a semântica do programa durante a compatibilização com IBT.

A Figura 5.4 mostra a distribuição do tamanho dos binários protegidos e desprotegidos no nível de otimização -O0. Nesse cenário, a média geométrica aumenta de aproximadamente 178,4 KB para 250,5 KB, correspondendo a um crescimento aproximado de 40,4%. Esse aumento é mais pronunciado em termos relativos porque os binários não otimizados avaliados possuem tamanho inicial menor e, ao mesmo tempo, preservam entradas de função mais regulares, o que favorece maior cobertura de instrumentação.

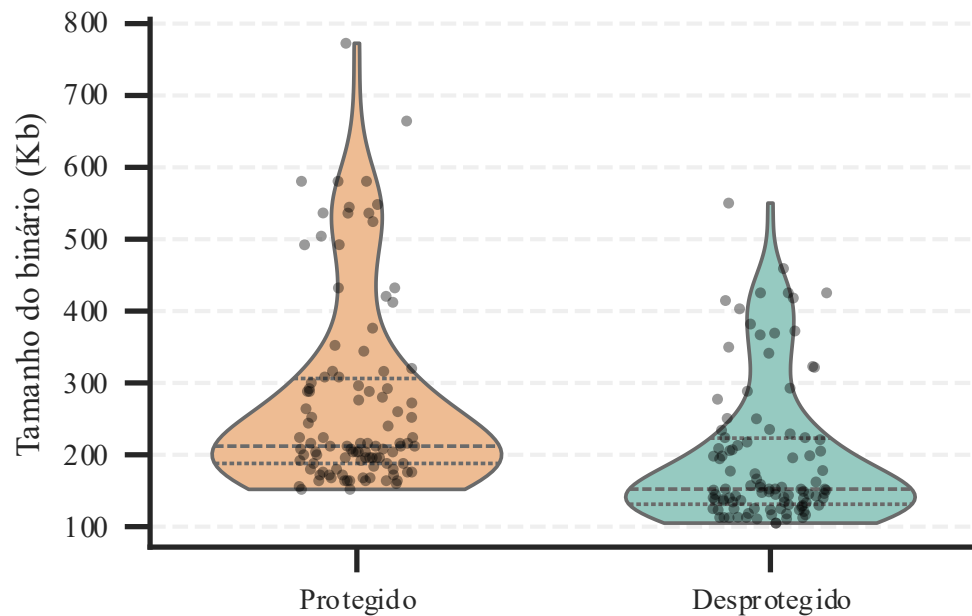


Figura 5.4 Distribuição do tamanho binário para os 102 programas do Coreutils em -O0.

A Figura 5.5 apresenta a distribuição de tamanho para os binários compilados com -O2. Nesse caso, a média geométrica aumenta de aproximadamente 338,0 KB para 407,4 KB. Embora o tamanho absoluto dos binários seja maior do que em -O0, o crescimento relativo tende a ser menor, pois parte do custo fixo introduzido pela instrumentação é diluído em executáveis maiores. Além disso, como uma fração das funções não pôde ser reescrita com *endbr64*, a quantidade total de material adicional inserido também é proporcionalmente menor.

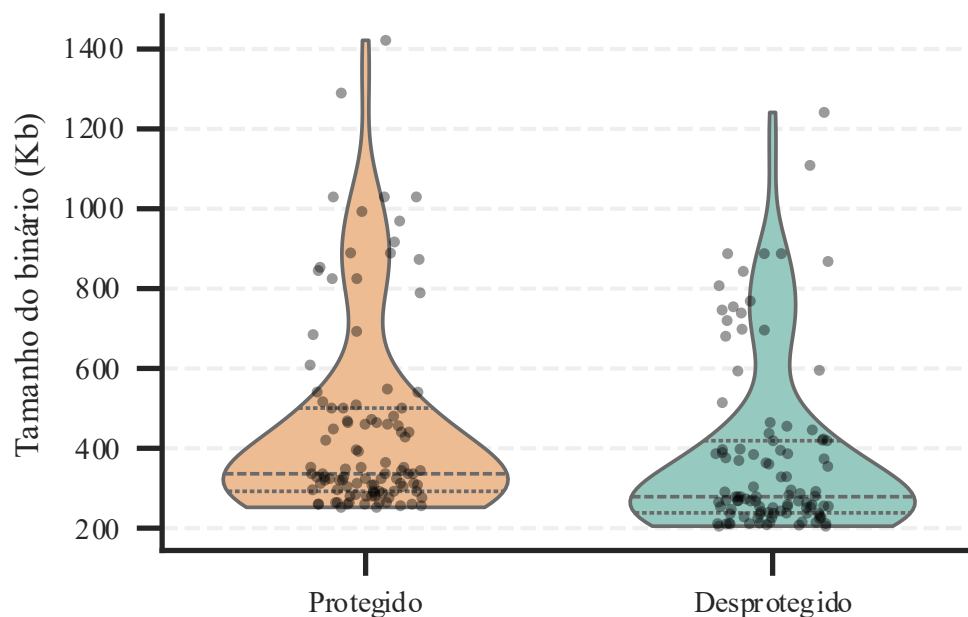


Figura 5.5 Distribuição do tamanho binário para os 102 programas do Coreutils em -O2.

A Figura 5.6 mostra o comportamento no nível -O3, em que a média geométrica aumenta de aproximadamente 414,3 KB para 502,8 KB. Assim como em -O2, os binários otimizados apresentam tamanho absoluto maior, mas o crescimento continua compatível com o tipo de transformação realizada. A distribuição confirma que a reescrita impõe um custo sistemático de espaço, mas não produz crescimento descontrolado nos executáveis avaliados.

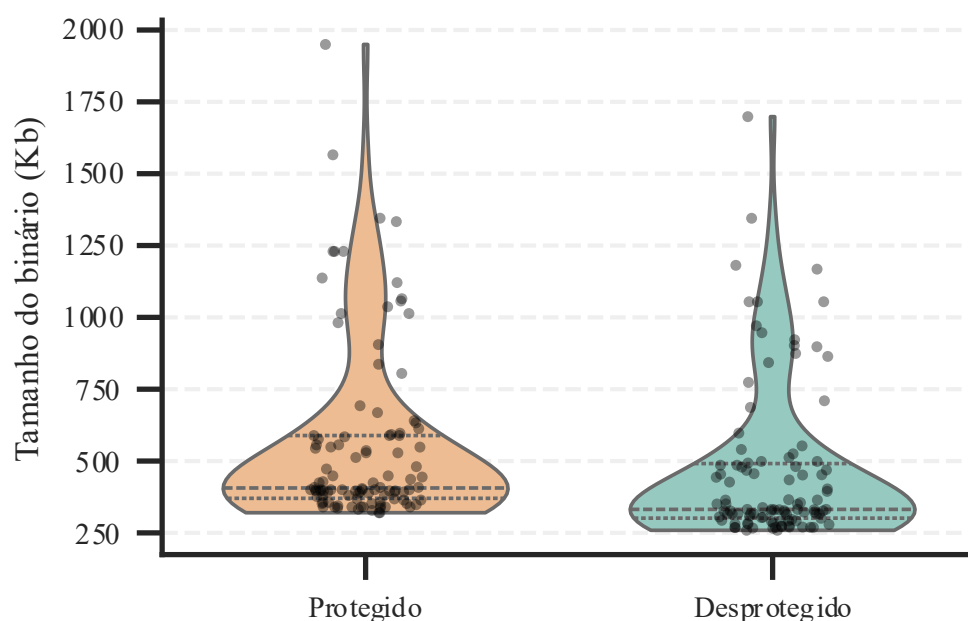


Figura 5.6 Distribuição do tamanho binário para os 102 programas do Coreutils em -O3.

Em conjunto, os resultados de tamanho mostram que o crescimento binário decorre diretamente da estratégia de compatibilização adotada. A inserção de marcadores, a materialização de trampolins e a preservação de instruções deslocadas exigem espaço adicional no binário final. Esse custo estrutural é esperado e deve ser avaliado em conjunto com os ganhos de compatibilidade e com a redução da superfície efetiva de ataque.

5.2.5 Redução da Superfície de Ataque

Para a análise do impacto da reescrita sobre a superfície efetiva de ataque disponível para técnicas de reutilização de código, utilizou-se a ferramenta ROPgadget em uma configuração agressiva, com `--depth 100` e `--multibr`, sem filtros adicionais. Essa configuração foi escolhida para favorecer uma enumeração sintática ampla de sequências semelhantes a *gadgets*, permitindo comparar o espaço bruto de *gadgets* com o conjunto efetivo de destinos preservados pela política de instrumentação.

A interpretação desses resultados exige uma distinção importante. O número bruto de *gadgets* reportado por uma ferramenta como ROPgadget não representa, por si só, a superfície efetiva de exploração sob uma política orientada a IBT. O ROPgadget realiza uma busca sintática, identificando sequências de instruções que correspondem a padrões de *gadgets*, mas não determina se essas sequências permanecem alcançáveis como destinos válidos de

transferências indiretas após a instrumentação. Em binários reescritos, o número bruto de *gadgets* pode inclusive aumentar, já que a transformação adiciona código executável, trampolins, blocos auxiliares e instruções realocadas.

A Figura 5.7 apresenta a distribuição de entradas de função recuperadas nos três níveis de otimização avaliados. Observa-se que o cenário -O0 tende a apresentar maior quantidade de entradas recuperadas por binário, enquanto -O2 e -O3 apresentam distribuições mais compactas. Esse comportamento é coerente com as transformações aplicadas por otimizações de compilador, que podem reduzir ou reorganizar funções, alterar prólogos e modificar a granularidade estrutural do código. Essas entradas são relevantes para a análise porque representam os principais candidatos à marcação com *endbr64* e, portanto, compõem parte do conjunto de destinos estruturalmente válidos após a instrumentação.

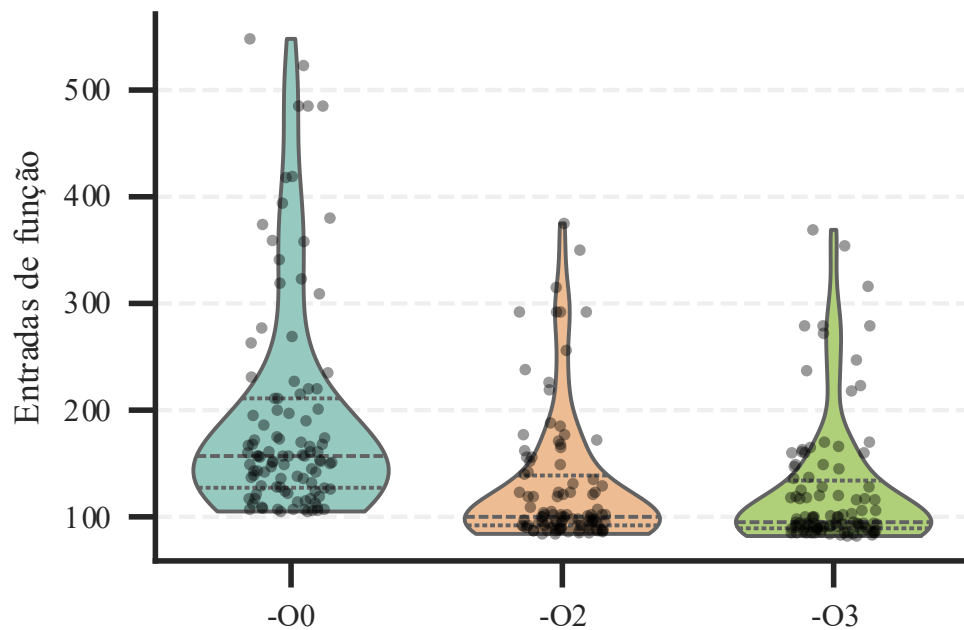


Figura 5.7 Distribuição de entradas de função para cada nível de otimização.

A Figura 5.8 mostra a distribuição de desvios indiretos identificados nos binários. Diferentemente das entradas de função, os desvios indiretos permanecem distribuídos em faixas semelhantes nos três níveis de otimização, embora com variações entre programas. Isso indica que a necessidade de tratar transferências indiretas não desaparece com a otimização e permanece relevante tanto em binários não otimizados quanto em layouts mais compactos. Esses desvios são importantes porque definem os pontos a partir dos quais um atacante poderia tentar redirecionar o fluxo de execução, respeitadas as restrições impostas pela política do IBT.

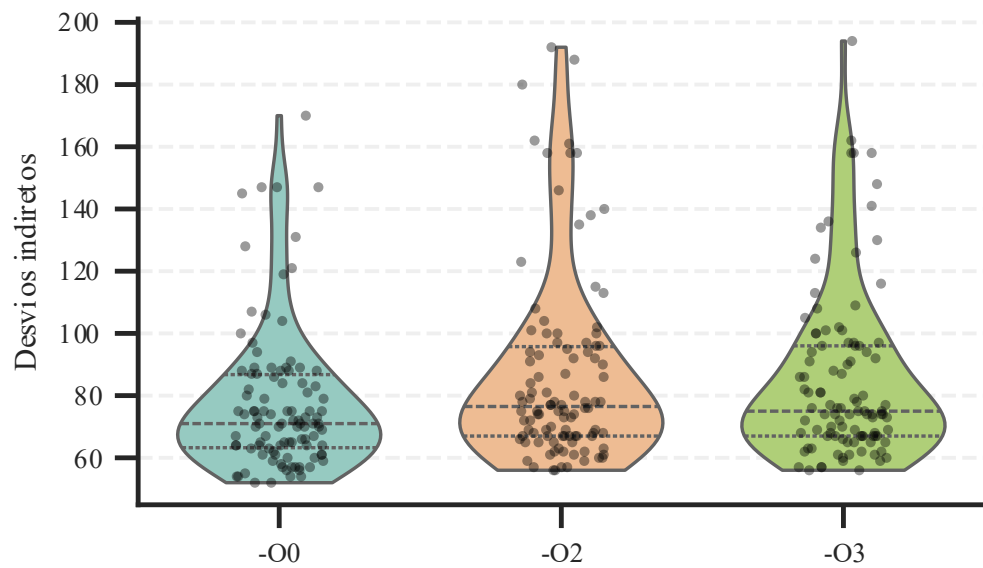


Figura 5.8 Distribuição de desvios indiretos para cada nível de otimização.

A Figura 5.9 apresenta a distribuição de *gadgets* sintáticos identificados pelo ROPgadget. Os valores permanecem elevados nos três níveis de otimização, o que confirma que o espaço sintático de sequências semelhantes a *gadgets* continua amplo mesmo após a aplicação da estratégia de compatibilização. Esse resultado não deve ser interpretado como falha da abordagem, pois a técnica proposta não tem como objetivo remover todos os *gadgets* do binário. Seu objetivo é reduzir o conjunto de destinos efetivamente alcançáveis por transferências indiretas sob a política adotada.

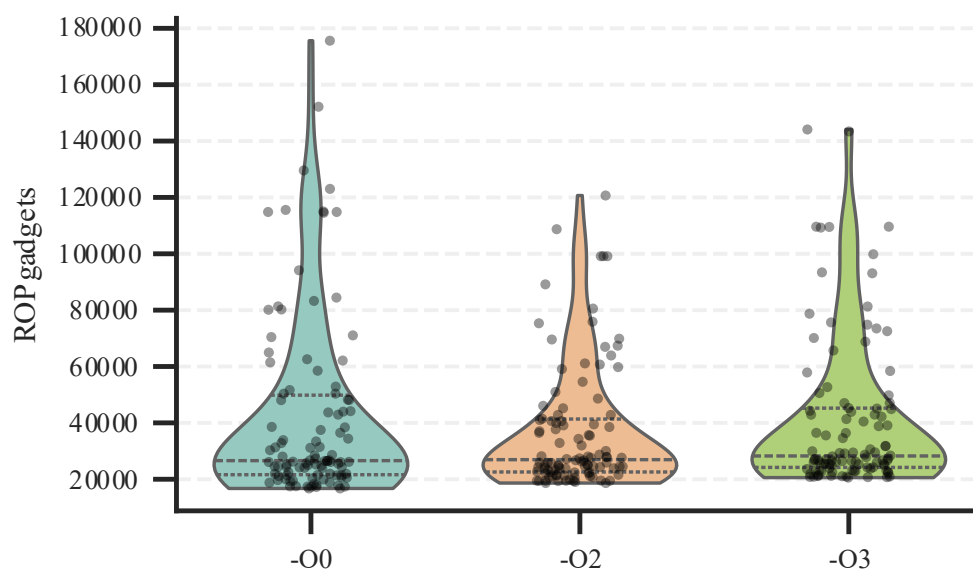


Figura 5.9 Distribuição de *gadgets* identificados pelo ROPgadget por nível de otimização.

A Figura 5.10 compara, em escala logarítmica, o número agregado de *gadgets* sintáticos, entradas de função e desvios indiretos nos três níveis de otimização. A diferença entre as barras evidencia que o espaço bruto reportado pelo ROPgadget é muito maior do que o conjunto de pontos efetivamente preservados como destinos válidos pela instrumentação. Os valores agregados mostram milhões de *gadgets* sintáticos, enquanto entradas de função e desvios indiretos permanecem em uma ordem de grandeza muito menor.

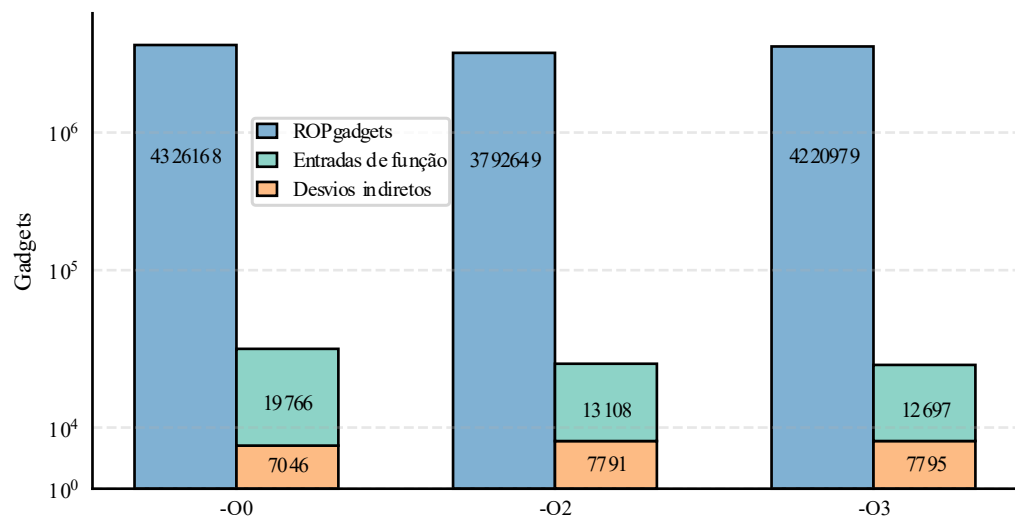


Figura 5.10 Comparação agregada entre o espaço sintático de *gadgets* e os alvos protegidos preservados pela política de instrumentação (em escala logarítmica).

A partir dessa comparação, os resultados indicam uma redução aproximada de 99,2% na superfície efetiva de transferência de controle, o que corresponde a cerca de 0,8% do espaço sintático original de *gadgets*. Esse resultado mostra que o efeito de segurança da abordagem não está em eliminar todas as sequências que uma ferramenta sintática pode classificar como gadget, mas em restringir quais destinos podem ser alcançados por transferências indiretas após a instrumentação.

Essa redução não significa que o IBT elimine toda possibilidade de exploração do fluxo de controle. Por construção, o IBT impõe uma política de granularidade grossa: ele impede desvios indiretos para locais que não iniciam com *endbr64*, mas ainda permite transferências para destinos estruturalmente válidos. Assim, um atacante que consiga redirecionar o fluxo para outra função legitimamente marcada ainda pode explorar a margem residual deixada pela política do próprio hardware. Esse limite não decorre de uma falha específica do BIN2CET, mas da semântica do mecanismo IBT. A contribuição do BIN2CET está em realizar a etapa estrutural necessária para que binários legados passem a respeitar essa política, reduzindo de forma expressiva o conjunto de destinos alcançáveis. Políticas mais finas, como verificações de tipo, aridade ou contexto de chamada, podem ser combinadas futuramente para restringir ainda mais os destinos válidos.

Essa distinção é fundamental para interpretar o impacto da proposta. A técnica não busca apagar todos os *gadgets* do binário, tarefa que seria incompatível com a preservação funcional de programas reais. Em vez disso, ela reduz o conjunto de pontos que permanecem

efetivamente utilizáveis como destinos de desvios indiretos. Essa interpretação está alinhada à ideia de que a viabilidade de ataques por reutilização de código depende não apenas da existência sintática de *gadgets*, mas da disponibilidade efetiva de caminhos que permitam alcançá-los durante a execução.

5.3 Considerações Finais

Este capítulo apresentou a avaliação experimental do BIN2CET em 102 programas da suíte GNU Coreutils, compilados como binários PIE sob diferentes níveis de otimização. Os resultados mostram que a abordagem proposta é capaz de compatibilizar binários ELF legados com os requisitos estruturais do IBT, alcançando cobertura completa das entradas de função recuperadas no cenário -O0 e cobertura superior a 80% em -O2 e -O3, indicando que a técnica continua aplicável em layouts otimizados, embora com limitações causadas por regiões de entrada mais compactas e restrições de posicionamento de trampolins.

Do ponto de vista funcional, os experimentos indicam que os binários instrumentados preservam o comportamento esperado nos testes executados. As violações observadas foram pontuais e associadas a situações específicas, como a chamada indireta da *libc* para funções *main* não reescritas ou a impossibilidade de posicionar trampolins em alguns saltos indiretos. Esses casos evidenciam os limites práticos da abordagem, mas também ajudam a delimitar com precisão as condições em que o *fallback* com *notrack* ou a ampliação do escopo de instrumentação podem ser necessários.

Os *overheads* de execução e tamanho mostram que a instrumentação impõe custos reais, mas compatíveis com o objetivo de adaptar binários já compilados a uma política de controle de fluxo mais restritiva. O aumento no número de instruções, nos desvios e no tamanho final dos binários é esperado, pois a transformação precisa inserir marcadores, redirecionar fluxo de controle e preservar instruções deslocadas. Assim, esses custos não devem ser interpretados como efeitos colaterais inesperados, mas como parte do custo estrutural da compatibilização sem recompilação.

Por fim, a análise da superfície de ataque reforça a diferença entre disponibilidade sintática de *gadgets* e alcançabilidade efetiva sob uma política de IBT. Mesmo que a reescrita introduza código adicional, o conjunto de destinos válidos para transferências indiretas passa a ser menor e explicitamente gerenciado. Desse modo, o BIN2CET contribui para reduzir a superfície efetiva disponível a ataques por reutilização de código, ao mesmo tempo em que preserva a funcionalidade dos programas avaliados.

Esses resultados mostram que a compatibilização proposta não deve ser interpretada como uma proteção absoluta contra todos os desvios maliciosos, mas como uma etapa estrutural necessária para que binários legados passem a operar sob a política do IBT. O BIN2CET reduz substancialmente o conjunto de destinos válidos para transferências indiretas e preserva a funcionalidade observada dos programas avaliados. As margens residuais, como destinos legitimamente marcados por *endbr64* ou pontos tratados com *notrack*, refletem compromissos inerentes entre segurança, compatibilidade e preservação semântica em binários já compilados. Essas margens podem ser reduzidas em trabalhos futuros por políticas complementares como a proposta do FineIBT.

6 Conclusões e Trabalhos Futuros

Este trabalho apresentou o BIN2CET, um sistema automatizado de reescrita binária estática voltado à compatibilização de binários ELF legados com o mecanismo *Indirect Branch Tracking* (IBT) da Intel CET. A proposta foi motivada pela dificuldade prática de recompilar softwares legados com suporte nativo a CET e pela necessidade de adaptar executáveis já compilados aos requisitos estruturais do IBT. Para isso, o BIN2CET combina análise binária, inserção de *endbr64* em alvos recuperados de transferências indiretas, preservação de instruções sobrescritas por meio de trampolins cientes de realocação e uso conservador de *notrack* nos casos em que a reescrita segura do destino não é possível.

A primeira questão de pesquisa investigou como compatibilizar binários ELF legados com os requisitos estruturais do IBT sem recompilação do código-fonte. Os resultados indicam que essa compatibilização pode ser alcançada por meio de reescrita binária estática, desde que a transformação seja capaz de inserir instruções *endbr64* em destinos recuperados e, ao mesmo tempo, preservar a semântica das instruções sobrescritas. O BIN2CET concretiza essa estratégia ao combinar análise binária, manipulação em nível de ELF, uso de trampolins cientes de realocação e tratamento conservador com *notrack* quando a reescrita segura do destino não é possível. Dessa forma, a abordagem evita depender da recuperação completa do grafo de fluxo de controle e permite adaptar binários já compilados aos requisitos do IBT sem acesso ao código-fonte.

A segunda questão de pesquisa analisou em que medida uma abordagem de reescrita binária estática baseada em microtransformações locais, trampolins e uso conservador de *notrack* preserva a funcionalidade dos executáveis e introduz overheads de desempenho e tamanho binário. A avaliação com os 102 programas da suíte GNU Coreutils mostrou que o BIN2CET alcançou cobertura completa das entradas de função recuperadas no cenário não otimizado e cobertura superior a 80% nos layouts otimizados. Os experimentos também indicaram que os binários transformados preservaram o comportamento funcional esperado nas cargas avaliadas, com violações pontuais associadas a limitações específicas de reescrita, como entradas de função compactas, restrições de posicionamento de trampolins e interações com chamadas indiretas realizadas pela *libc*. As medições de desempenho indicaram *overhead* ponderado de execução entre aproximadamente 22% para -O0 e em torno de 42% para -O2 e -O3 nos principais contadores de tempo, enquanto a análise por programa evidenciou variação entre os binários avaliados, com dispersão reportada por meio de desvio padrão, intervalo de confiança de 95%, mínimo e máximo. O crescimento do tamanho binário refletiu o custo

estrutural esperado da inserção de instruções *endbr64*, redirecionamentos, trampolins e instruções realocadas. A validação em tempo de execução com PIN, nesse contexto, foi utilizada como evidência complementar para observar se chamadas e saltos indiretos alcançavam destinos compatíveis com a política adotada, sem substituir a estratégia principal de compatibilização, que permanece estática.

A terceira questão de pesquisa avaliou em que medida o BIN2CET reduz a superfície efetiva de ataque associada a transferências indiretas válidas pela política do IBT em binários legados instrumentados. Os resultados mostram que a contribuição da metodologia não deve ser medida apenas pelo número bruto de *gadgets* sintáticos reportados por ferramentas como ROPgadget, mas pelo conjunto efetivo de destinos válidos para transferências indiretas. Embora a reescrita possa introduzir novas sequências sintáticas, especialmente em razão dos trampolins e das instruções realocadas, a política do IBT restringe a alcançabilidade efetiva aos destinos compatíveis com *endbr64* ou explicitamente tratados com *notrack*. Sob essa interpretação, o BIN2CET reduziu a superfície efetiva associada a transferências indiretas em aproximadamente 99,2%, deixando como alcançável cerca de 0,8% do espaço sintático original de *gadgets*.

De modo geral, os resultados obtidos confirmam a viabilidade técnica da compatibilização de binários legados com IBT por meio de reescrita binária estática. A abordagem mostrou-se especialmente efetiva em binários que preservam prólogos regulares de função, como observado no cenário -O0, mas também permaneceu aplicável em binários otimizados, apesar das limitações impostas por entradas de função mais compactas e restrições de posicionamento de trampolins. A execução bem-sucedida dos binários transformados nas cargas avaliadas fornece evidências de preservação funcional, enquanto os resultados de superfície de ataque mostram que a política implementada reduz de forma expressiva o conjunto de destinos efetivamente alcançáveis por desvios indiretos.

As limitações identificadas também ajudam a delimitar o escopo da proposta. Em binários otimizados, algumas funções não puderam ser reescritas porque a região inicial não oferecia espaço suficiente para acomodar a instrumentação sem violar fronteiras de instrução. Em outros casos, o E9Patch não encontrou uma posição adequada para materializar o trampolim correspondente. Também foram observadas situações específicas em que funções *main* não reescritas foram alcançadas por chamadas indiretas realizadas pela *libc*, o que indica que a ampliação do escopo de instrumentação para bibliotecas compartilhadas pode ser necessária em determinados cenários. Além disso, o uso de *notrack*, embora importante como mecanismo de compatibilidade, é menos restritivo do ponto de vista de segurança do que a marcação explícita do destino com *endbr64*, razão pela qual deve permanecer como *fallback* conservador.

Como trabalhos futuros, pretende-se ampliar a avaliação para pilhas de software mais complexas e próximas de ambientes de produção, incluindo serviços como Nginx vinculado ao OpenSSL. Essa ampliação permitirá observar o comportamento do BIN2CET em aplicações de longa execução, com cargas de trabalho mais realistas e padrões de fluxo de controle distintos daqueles encontrados em utilitários de linha de comando. Também se pretende melhorar os mecanismos de análise para binários sem símbolos, reduzir a necessidade de marcações conservadoras com *notrack* e aprimorar o tratamento de instruções sensíveis à realocação em layouts mais complexos.

Outra direção relevante consiste em comparar o BIN2CET com outras abordagens de proteção binária e investigar extensões capazes de impor políticas mais finas sobre destinos de transferências indiretas. Pretende-se também investigar a extensão do BIN2CET para cenários com geração dinâmica de código, incluindo ambientes com JIT *compilation*, que é uma técnica que traduz o código de um programa em código de máquina nativo em tempo de execução, em vez de fazer isso antes do programa rodar. Essa extensão exigirá integrar a compatibilização com IBT a mecanismos de carga, tradução ou instrumentação dinâmica, de modo que regiões de código produzidas em tempo de execução também possam receber marcações compatíveis com a política do IBT. Também pretende-se ampliar a automatização das decisões de instrumentação, incluindo a seleção automática entre normalização com *endbr64* e tratamento conservador com *notrack*, a escolha de locais seguros para trampolins e a geração de relatórios explicativos para pontos não instrumentados.

Embora o foco desta dissertação tenha sido a compatibilização estrutural com IBT, trabalhos como FineIBT indicam que há espaço para mecanismos adicionais baseados em tipo, aridade ou contexto de chamada. Explorar tais políticas diretamente sobre binários já compilados pode ampliar as garantias de segurança oferecidas pela abordagem, mantendo a vantagem de não depender de recompilação ou acesso ao código-fonte.

Por fim, a disponibilização de um artefato reproduzível com scripts, ambiente Docker e dados experimentais constitui parte importante da contribuição deste trabalho. Esse material permite que outros pesquisadores reproduzam os resultados, inspecionem as decisões de instrumentação e estendam o BIN2CET para novos cenários. Assim, além de demonstrar a viabilidade da compatibilização com IBT em binários legados, esta dissertação oferece uma base prática para investigações futuras em reescrita binária, endurecimento de software e aplicação de mecanismos modernos de proteção a sistemas já implantados.

6.1 Desafios encontrados

O desenvolvimento desta pesquisa evidenciou que a compatibilização de binários legados com o mecanismo IBT envolve desafios técnicos que vão além da simples inserção de instruções *endbr64*. O primeiro desafio está relacionado à própria natureza dos binários já compilados. Diferentemente de um programa em código-fonte, no qual o compilador pode reorganizar instruções e inserir marcadores de forma controlada, um binário possui um layout fixo, com instruções já posicionadas em endereços específicos. Qualquer modificação precisa respeitar fronteiras de instrução, permissões de segmento, alinhamento e dependências de posição. Assim, a inserção de *endbr64* em um alvo de desvio indireto pode exigir o deslocamento de instruções originalmente presentes naquela região, o que torna indispensável o uso de trampolins e mecanismos de preservação semântica.

Outro desafio importante decorre das instruções dependentes de posição, especialmente aquelas que utilizam endereçamento relativo a RIP. Quando uma instrução desse tipo é deslocada para um trampolim, seus operandos não podem ser simplesmente copiados sem alteração, pois o deslocamento original passa a ser interpretado a partir de uma nova posição no código. Caso essa diferença não seja corrigida, a instrução pode acessar uma variável global ou estrutura de dados incorreta, preservando aparentemente o fluxo de controle, mas corrompendo a semântica do programa. Por essa razão, foi necessário incorporar ao BIN2CET suporte à correção de referências relativas a RIP, de modo que instruções realocadas continuem acessando os mesmos alvos semânticos do binário original.

Os experimentos realizados nesta dissertação não contemplam cenários com compilação JIT, código auto-modificável ou geração dinâmica de código em tempo de execução. A abordagem proposta atua de forma estática sobre o binário ELF em disco, antes da execução, e pressupõe que as regiões de código a serem instrumentadas estejam presentes no artefato analisado. Portanto, aplicações que geram código dinamicamente, como máquinas virtuais, navegadores com motores *JavaScript* ou *runtimes* com JIT, exigem mecanismos adicionais de instrumentação em tempo de carga ou execução, estando fora do escopo experimental deste trabalho.

Também se observou uma dificuldade associada aos diferentes níveis de otimização utilizados na compilação dos programas avaliados. No cenário -O0, as funções tendem a preservar prólogos mais regulares e espaços de entrada mais previsíveis, o que favorece a inserção de *endbr64* e a criação de redirecionamentos seguros. Em -O2 e -O3, entretanto, o compilador produz layouts mais compactos e frequentemente reduz a disponibilidade de bytes

consecutivos no início das funções. Essa compactação dificulta a reescrita segura de alguns alvos, pois a instrumentação não pode violar fronteiras de instrução nem sobrescrever trechos sem garantir sua reconstrução correta em trampolins. Como consequência, parte das funções em binários otimizados não pôde ser tratada pela estratégia preferencial baseada em *endbr64*.

Notou-se ainda um obstáculo no momento de avaliar o nível de otimização -O1. Embora esse nível representasse um ponto intermediário relevante entre o cenário não otimizado e os níveis mais agressivos, não foi possível consolidar uma avaliação estável para essa configuração dentro do *pipeline* experimental. Observou-se durante os testes com -O1 que os binários não possuíam o espaço mínimo de 5 bytes para a inserção do desvio para o trampolim, de forma que, ao tentar acomodar o desvio era-se cortado a próxima instrução no meio, podendo comprometer a semântica do programa. Com as restrições de posicionamento de trampolins, resultou-se em falhas que impediram a obtenção de uma base comparável às configurações -O0, -O2 e -O3. Por essa razão, os experimentos finais foram concentrados nos três cenários em que o processo pôde ser reproduzido de forma consistente. Essa limitação não invalida os resultados obtidos, mas indica que a análise de -O1 permanece como um ponto a ser retomado em trabalhos futuros, especialmente para entender quais características específicas desse nível de otimização dificultam a aplicação estável da metodologia.

A validação em tempo de execução também apresentou limitações. O verificador baseado em PIN foi fundamental para observar se chamadas e saltos indiretos alcançavam destinos iniciados por *endbr64* ou tratados com *notrack*. No entanto, alguns programas e testes da suíte Coreutils apresentaram incompatibilidades com PIN, exigindo sua remoção dessa etapa da avaliação. Além disso, foram observados casos específicos em que funções *main* não reescritas foram alcançadas por chamadas indiretas realizadas pela *libc*, gerando violações fora do escopo direto da instrumentação aplicada aos binários avaliados. Esses casos indicam que, em determinados cenários, a compatibilização completa pode exigir ampliar o escopo de análise e transformação para bibliotecas compartilhadas ou componentes externos ao executável principal.

Por fim, a consolidação do BIN2CET como ferramenta automatizada exigiu integrar diferentes etapas em um fluxo único de análise, geração de artefatos, transformação e validação. Essa integração envolveu o uso combinado dos analisadores *objdump* e *Ghidra*, manipulação ELF com LIEF, extensão do *E9Patch* para suporte à instrumentação desejada, geração de arquivos intermediários e validação dos binários instrumentados. O maior obstáculo consistiu em transformar uma metodologia inicialmente exploratória em um *pipeline* reproduzível e avaliável em escala. Superar essa etapa permitiu que a pesquisa deixasse de depender de estudos

de caso pontuais e passasse a ser avaliada sobre um conjunto amplo de programas reais. Ainda assim, enfrentar essas dificuldades foi fundamental para que a metodologia proposta evoluísse de um conjunto de ideias conceituais para um fluxo concreto de instrumentação aplicado a um conjunto de binários reais, com evidências da preservação funcional e compatibilização estrutural com os requisitos da Intel CET.

Referências

- ABADI, Martín; BUDIU, Mihai; ERLINGSSON, Úlfar; LIGATTI, Jay. Control-flow integrity. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 12., 2005, Alexandria. Proceedings [...]. New York: ACM, 2005. p. 340–353.
- ALEPH ONE. Smashing the stack for fun and profit. Phrack Magazine, v. 7, n. 49, art. 14, 1996. Disponível em: <https://phrack.org/issues/49/14>. Acesso em: 3 ago. 2025.
- BARBERIS, Enrico; FRIGO, Pietro; MUENCH, Marius; BOS, Herbert; GIUFFRIDA, Cristiano. Branch History Injection: on the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In: USENIX SECURITY SYMPOSIUM, 31., 2022, Boston. Proceedings [...]. Berkeley: USENIX Association, 2022. p. 971–988.
- BAUMANN, Jonathan; KIM, Yonghyun; FARBA, Yan; HRITCU, Catalin; LEATHERMAN-BROOKS, Julay. Trio securis: formally verified protection against speculative control-flow hijacking. arXiv, 2026. Disponível em: <https://arxiv.org/abs/2601.22978>. Acesso em: 30 jun. 2026.
- BERNAT, Andrew R.; MILLER, Barton P. Anywhere, any-time binary instrumentation. In: ACM SIGPLAN-SIGSOFT WORKSHOP ON PROGRAM ANALYSIS FOR SOFTWARE TOOLS AND ENGINEERING, 10., 2011, Szeged. Proceedings [...]. New York: Association for Computing Machinery, 2011. p. 9–16. DOI: <https://doi.org/10.1145/2024569.2024572>.
- BLETSCHE, Tyler; JIANG, Xuxian; FREEH, Vince W.; LIANG, Zhenkai. Jump-Oriented Programming: a new class of code-reuse attack. In: ACM SYMPOSIUM ON INFORMATION, COMPUTER AND COMMUNICATIONS SECURITY, 6., 2011, Hong Kong. Proceedings [...]. New York: Association for Computing Machinery, 2011. p. 30–40. DOI: <https://doi.org/10.1145/1966913.1966919>.
- CANIATO, Bruno. Vazamentos de dados no governo crescem mais de 20 vezes em 4 anos. Veja, São Paulo, 28 jan. 2025. Disponível em: <https://veja.abril.com.br/coluna/maquiavel/vazamentos-de-dados-no-governo-crescem-mais-de-20-vezes-em-4-anos/>. Acesso em: 14 nov. 2025.
- CARLINI, Nicholas; WAGNER, David. ROP is still dangerous: breaking modern defenses. In: USENIX SECURITY SYMPOSIUM, 23., 2014, San Diego. Proceedings [...]. Berkeley: USENIX Association, 2014.
- D’ELIA, Daniele Cono; INVIDIA, Lorenzo; PALMARO, Federico; QUERZONI, Leonardo. Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. Digital Threats: Research and Practice, New York, v. 3, n. 2, 2022. DOI: <https://doi.org/10.1145/3478520>.
- DINESH, Sushant; BUROW, Nathan; XU, Dongyan; PAYER, Mathias. RetroWrite: statically instrumenting COTS binaries for fuzzing and sanitization. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 2020, San Francisco. Proceedings [...]. Piscataway: IEEE, 2020. p. 1497–1511.

DUCK, Gregory J.; GAO, Xiang; ROYCHOUDHURY, Abhik. Binary rewriting without control flow recovery. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 41., 2020, London. Proceedings [...]. New York: Association for Computing Machinery, 2020. p. 151–163. DOI: <https://doi.org/10.1145/3385412.3385972>.

DYNINST PROJECT. Dyninst 12.1.0 documentation. Disponível em: <https://researchapps.github.io/dyninst-docs/>. Acesso em: 5 jun. 2025.

GAIDIS, Alexander J.; MOREIRA, Joao; SUN, Ke; MILBURN, Alyssa; ATLIDAKIS, Vaggelis; KEMERLIS, Vasileios P. FineIBT: fine-grain control-flow enforcement with Indirect Branch Tracking. In: INTERNATIONAL SYMPOSIUM ON RESEARCH IN ATTACKS, INTRUSIONS AND DEFENSES, 26., 2023, Hong Kong. Proceedings [...]. New York: Association for Computing Machinery, 2023. p. 527–546. DOI: <https://doi.org/10.1145/3607199.3607219>.

GUO, Wenbo; MU, Dongliang; XING, Xinyu; DU, Min; SONG, Dawn. DEEPVSA: facilitating value-set analysis with deep learning for postmortem program analysis. In: USENIX SECURITY SYMPOSIUM, 28., 2019, Santa Clara. Proceedings [...]. Berkeley: USENIX Association, 2019. p. 1787–1804.

HAWKINS, William H.; HISER, Jason D.; CO, Michele; NGUYEN-TUONG, Anh; DAVIDSON, Jack W. Zipr: efficient static binary rewriting for security. In: ANNUAL IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 47., 2017, Denver. Proceedings [...]. Piscataway: IEEE, 2017. p. 559–566.

HISER, Jason; NGUYEN-TUONG, Anh; HAWKINS, William; MCGILL, Matthew; CO, Michele; DAVIDSON, Jack. Zipr++: exceptional binary rewriting. In: WORKSHOP ON FORMING AN ECOSYSTEM AROUND SOFTWARE TRANSFORMATION, 2017, Dallas. Proceedings [...]. New York: Association for Computing Machinery, 2017. p. 9–15.

HP. BIOS de computadores HP: atualizações de segurança de maio de 2024 para potencial estouro de buffer de pilha. HP Customer Support, 2024. Disponível em: https://support.hp.com/br-pt/document/ish_10737448-10738512-16/hpsbhf03943. Acesso em: 3 ago. 2025.

INTEL CORPORATION. Control-flow Enforcement Technology specification. Santa Clara: Intel, 2019. Disponível em: <https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf>. Acesso em: 20 nov. 2024.

KIM, Hyungseok; KIM, Soomin; CHA, Sang Kil. Towards sound reassembly of modern x86-64 binaries. In: ACM INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 30., 2025, Rotterdam. Proceedings [...]. New York: Association for Computing Machinery, 2025. v. 2, p. 1317–1333. DOI: <https://doi.org/10.1145/3676641.3716026>.

MALVICA, Marco. Bypassing Intel CET with counterfeit objects. Offensive Security Blog, 25 ago. 2022. Disponível em: <https://www.offsec.com/blog/bypassing-intel-cet-with-counterfeit-objects/>. Acesso em: 23 fev. 2025.

MENG, Xiaozhu; CHAMITH, Buddhika C.; NEWTON, Ryan. ShadowGuard: optimizing the policy and mechanism of Shadow Stack instrumentation using binary static analysis. arXiv, 2021. Disponível em: <https://arxiv.org/abs/2002.07748>. Acesso em: 30 jun. 2026.

MENG, Xiaozhu; LIU, Weijie. Incremental CFG patching for binary rewriting. In: ACM INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 26., 2021, Virtual. Proceedings [...]. New York: Association for Computing Machinery, 2021. DOI: <https://doi.org/10.1145/3445814.3446765>.

MENG, Xiaozhu; MILLER, Barton P. Binary code is not easy. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 25., 2016, Saarbrücken. Proceedings [...]. New York: Association for Computing Machinery, 2016.

ORMAN, Hilarie K. The Morris worm: a fifteen-year perspective. IEEE Security & Privacy, Piscataway, v. 1, n. 5, p. 35–43, 2003. DOI: <https://doi.org/10.1109/MSECP.2003.1236233>.

PAX TEAM. RAP: RIP ROP. H2HC, 2015. Disponível em: <https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf>. Acesso em: 20 jul. 2026.

RIMSA, Andrei; AMARAL, José Nelson; QUINTÃO, Fernando Magno. Efficient and precise dynamic construction of control flow graphs. In: BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 23., 2019, Salvador. Proceedings [...]. New York: Association for Computing Machinery, 2019.

SCHULTE, Eric; BROWN, Michael D.; FOLTS, Vlad. A broad comparative evaluation of x86-64 binary rewriters. In: WORKSHOP ON CYBER SECURITY EXPERIMENTATION AND TEST, 15., 2022. Proceedings [...]. Berkeley: USENIX Association, 2022.

SHACHAM, Hovav. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 14., 2007, Alexandria. Proceedings [...]. New York: Association for Computing Machinery, 2007. p. 552–561. DOI: <https://doi.org/10.1145/1315245.1315313>.

SHANBHOGUE, Vedvyas; GUPTA, Deepak; SAHITA, Ravi. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In: INTERNATIONAL WORKSHOP ON HARDWARE AND ARCHITECTURAL SUPPORT FOR SECURITY AND PRIVACY, 8., 2019, Phoenix. Proceedings [...]. New York: Association for Computing Machinery, 2019.

SMITHSON, Matthew; ELWAZEER, Khaled; ANAND, Kapil; KOTHA, Aparna; BARUA, Rajeev. Static binary rewriting without supplemental information: overcoming the tradeoff between coverage and correctness. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 20., 2013, Koblenz. Proceedings [...]. Piscataway: IEEE, 2013.

STACKPOLE, Beth. MIT report details new cybersecurity risks. Ideas Made to Matter, Cambridge, abr. 2024. Disponível em: <https://mitsloan.mit.edu/ideas-made-to-matter/mit-report-details-new-cybersecurity-risks>. Acesso em: 12 nov. 2025.

TYMBURIBÁ, M.; SOUSA, H. A. de; PEREIRA, F. M. Q. Multilayer ROP protection via microarchitectural units available in commodity hardware. In: ANNUAL IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 49., 2019, Portland. Proceedings [...]. Piscataway: IEEE, 2019.

VAN PUT, Ludo; CHANET, Dominique; DE BUS, Bruno; DE SUTTER, Bjorn; DE BOSSCHE, Koen de. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In: IEEE INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND INFORMATION TECHNOLOGY, 2005, Athens. Proceedings [...]. Piscataway: IEEE, 2005. p. 7–12.

WANG, Ruoyu; SHOSHITAISHVILI, Yan; BIANCHI, Antonio; MACHIRY, Aravind; GROSEN, John; GROSEN, Paul; KRUEGEL, Christopher; VIGNA, Giovanni. Ramblr: making reassembly great again. In: NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM, 2017, San Diego. Proceedings [...]. Reston: Internet Society, 2017.

WANG, Shuai; WANG, Pei; WU, Dinghao. UROBOROS: instrumenting stripped binaries with static reassembling. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING, 23., 2016, Osaka. Proceedings [...]. Piscataway: IEEE, 2016.

WENZL, Matthias; MERZDOVNIK, Georg; ULLRICH, Johanna; WEIPPL, Edgar. From hack to elaborate technique: a survey on binary rewriting. ACM Computing Surveys, New York, v. 52, n. 3, art. 49, 2019. DOI: <https://doi.org/10.1145/3316415>.

WIEBING, Sander; DE FAVERI TRON, Alvis; BOS, Herbert; GIUFFRIDA, Cristiano. InSpectre Gadget: inspecting the residual attack surface of cross-privilege Spectre v2. In: USENIX SECURITY SYMPOSIUM, 33., 2024, Philadelphia. Proceedings [...]. Berkeley: USENIX Association, 2024.

XIE, Mengyao; WU, Chenggang; ZHANG, Yinqian; XU, Jiali; LAI, Yuanming; KANG, Yan; WANG, Wei; WANG, Zhe. CETIS: retrofitting Intel CET for generic and efficient intra-process memory isolation. In: ACM SIGSAC CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 2022, Los Angeles. Proceedings [...]. New York: Association for Computing Machinery, 2022. p. 2989–3002. DOI: <https://doi.org/10.1145/3548606.3559344>.

ZHANG, Mingwei; QIAO, Rui; HASABNIS, Niranjan; SEKAR, R. A platform for secure static binary instrumentation. In: ACM SIGPLAN/SIGOPS INTERNATIONAL CONFERENCE ON VIRTUAL EXECUTION ENVIRONMENTS, 10., 2014, Salt Lake City. Proceedings [...]. New York: Association for Computing Machinery, 2014.

ZHAO, Brian; YANG, Yiwei; ZHENG, Yusheng; QUINN, Andi. Exploiting Control-flow Enforcement Technology for sound and precise static binary disassembly. arXiv, 2025. Disponível em: <https://arxiv.org/abs/2506.09426>. Acesso em: 30 jun. 2026.

Apêndice A

Este apêndice apresenta os overheads de desempenho calculados por programa para os níveis de otimização -O0, -O2 e -O3. Os valores foram obtidos a partir dos CSVs consolidados de desempenho, comparando a versão instrumentada pelo BIN2CET com a versão original de cada programa. Esses dados complementam a análise agregada apresentada no Capítulo 5.2.3 e permitem observar a variação do custo da instrumentação entre os 98 binários avaliados em cada nível de otimização.

Tabela A.1 Overhead de desempenho por programa para cada nível de otimização.

Otimização	-O0	-O0	-O0	-O0	-O0
Programa	l	b2sum	base32	base64	basename
Cache-Misses Overhead (%)	14.75	17.12	15.13	16.0	15.01
Cache-Misses Protegido	2874.61	3578.72	3706.06	3194.52	2983.35
Cache-Misses Desprotegido	2505.03	3055.7	3218.98	2753.86	2594.03
Cache-References Overhead (%)	25.18	23.2	22.45	24.4	20.82
Cache-References Protegido	23918.08	27132.8	26943.45	24746.68	23467.33
Cache-References Desprotegido	19107.46	22023.16	22003.95	19892.27	19423.13
Branch-Misses Overhead (%)	36.75	28.43	34.51	33.59	26.96
Branch-Misses Protegido	11269.05	11709.35	12010.56	11284.88	10258.34
Branch-Misses Desprotegido	8240.58	9117.64	8929.45	8447.47	8080.18
Branches Overhead (%)	40.07	30.61	39.05	36.1	25.86
Branches Protegido	230444.97	237446.55	237157.37	224501.94	198622.6
Branches Desprotegido	164518.67	181793.03	170551.53	164954.34	157809.04
Stalled-Cycles-Frontend Overhead (%)	28.48	26.68	28.83	29.89	23.32
Stalled-Cycles-Frontend Protegido	1399996.5	1552951.7	1557736.9	1440202.8	1323668.6
Stalled-Cycles-Frontend Desprotegido	1089681.5	1225840.3	1209101.2	1108759.5	1073359.3
Task-Clock Overhead (%)	34.55	30.0	35.71	34.55	24.53
Task-Clock Protegido	0.74	0.78	0.76	0.74	0.66
Task-Clock Desprotegido	0.55	0.6	0.56	0.55	0.53
Cycles Overhead (%)	33.22	28.67	32.69	32.91	25.17
Cycles Protegido	2016905.5	2187984.7	2197392.7	2044609.4	1855720.4
Cycles Desprotegido	1513912.6	1700414.4	1656051.7	1538373.3	1482598.2
Instructions Overhead (%)	42.22	31.13	39.63	37.67	27.21
Instructions Protegido	1272083.3	1309619.2	1311351.7	1244341.5	1097559.3
Instructions Desprotegido	894450.21	998748.91	939133.76	903831.35	862825.81

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	basenc	cat	chgrp	chmod	chown	chroot	cksum	comm
Cache-Misses Overhead (%)	16.71	13.52	13.71	16.55	15.41	19.09	3.36	14.41
Cache-Misses Protegido	3819.46	4223.13	4804.5	3929.67	4015.4	3123.16	21131.58	3408.71
Cache-Misses Desprotegido	3272.52	3720.2	4225.3	3371.71	3479.21	2622.48	20444.5	2979.4
Cache-References Overhead (%)	23.87	20.45	18.76	23.07	20.49	27.2	8.82	21.05
Cache-References Protegido	27649.66	27487.79	34066.88	29013.69	30259.71	25017.26	69983.06	26100.74
Cache-References Desprotegido	22322.29	22820.13	28684.54	23574.57	25114.12	19668.07	64308.18	21562.61
Branch-Misses Overhead (%)	39.42	31.68	27.03	30.9	30.56	31.73	20.79	24.0
Branch-Misses Protegido	12636.87	11358.94	15300.48	12591.05	13962.21	11131.71	21002.25	11134.75
Branch-Misses Desprotegido	9064.02	8626.41	12045.12	9618.95	10694.49	8450.66	17387.22	8979.88
Branches Overhead (%)	48.6	31.64	30.63	38.16	34.55	31.36	15.02	26.17
Branches Protegido	267334.46	238895.83	314782.76	257117.43	286700.91	217447.67	699785.45	216056.45
Branches Desprotegido	179904.82	181483.47	240967.87	186101.31	213079.82	165529.93	608417.6	171248.61
Stalled-Cycles-Frontend Overhead (%)	34.64	23.85	23.98	28.99	26.87	22.32	12.92	23.56
Stalled-Cycles-Frontend Protegido	1675130.6	1610754.4	1983714.3	1699901.8	1786651.1	1397732.7	4036662.8	1471896.1
Stalled-Cycles-Frontend Desprotegido	1244193.7	1300601.2	1599969.9	1317876.3	1408290.7	1142679.9	3574928.8	1191239.9
Task-Clock Overhead (%)	36.67	26.98	29.11	32.31	30.0	23.73	14.36	18.33
Task-Clock Protegido	0.82	0.8	1.02	0.86	0.91	0.73	2.07	0.71
Task-Clock Desprotegido	0.6	0.63	0.79	0.65	0.7	0.59	1.81	0.6
Cycles Overhead (%)	39.7	26.82	26.96	32.47	30.24	25.74	14.13	25.19
Cycles Protegido	2392338.9	2257856.7	2818055.4	2397257.3	2551544.3	1979838.4	5805048.0	2053659.0
Cycles Desprotegido	1712453.1	1780343.0	2219587.4	1809674.5	1959145.3	1574520.8	5086497.6	1640454.7
Instructions Overhead (%)	49.78	32.62	31.96	39.45	35.92	32.13	15.49	27.1
Instructions Protegido	1480196.3	1321747.1	1735524.9	1428293.3	1584775.4	1202247.2	3812856.9	1195660.0
Instructions Desprotegido	988275.13	996634.68	1315158.4	1024210.1	1165965.5	909890.69	3301329.2	940692.25

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	cp	csplit	cut	date	dd	df	dir	dircolors
Cache-Misses Overhead (%)	15.58	16.52	18.0	17.7	17.96	16.22	21.29	11.47
Cache-Misses Protegido	5184.19	4163.29	3601.07	3931.15	4232.28	3785.43	4035.59	3201.37
Cache-Misses Desprotegido	4485.51	3573.12	3051.64	3339.92	3588.0	3257.06	3327.18	2872.08
Cache-References Overhead (%)	22.39	21.56	22.27	25.25	23.21	22.87	25.74	21.44
Cache-References Protegido	32862.29	32477.62	30783.45	30801.31	29474.95	27893.3	28068.41	25393.9
Cache-References Desprotegido	26850.42	26716.89	25176.09	24591.17	23923.33	22700.81	22322.69	20911.23
Branch-Misses Overhead (%)	38.64	36.25	34.53	35.07	34.1	29.15	44.83	25.94
Branch-Misses Protegido	14918.34	14128.31	13431.27	14083.1	12916.84	13087.93	14073.25	10975.18
Branch-Misses Desprotegido	10760.65	10369.44	9984.12	10426.8	9632.28	10134.09	9717.41	8714.81
Branches Overhead (%)	48.18	44.08	37.78	44.94	40.76	35.04	55.39	28.86
Branches Protegido	320042.2	298259.84	290846.34	290340.97	266907.18	285360.4	300523.61	219466.02
Branches Desprotegido	215981.71	207012.89	211101.04	200312.47	189615.87	211319.07	193402.82	170310.81
Stalled-Cycles-Frontend Overhead (%)	32.57	31.27	29.56	34.04	29.96	29.18	39.88	24.49
Stalled-Cycles-Frontend Protegido	2039858.7	1888851.3	1761481.0	1818365.7	1746343.9	1667724.6	1747989.6	1425909.6
Stalled-Cycles-Frontend Desprotegido	1538756.8	1438958.9	1359549.2	1356628.4	1343760.5	1291034.1	1249644.0	1145415.0
Task-Clock Overhead (%)	37.84	38.57	32.84	37.88	32.31	30.77	50.0	26.32
Task-Clock Protegido	1.02	0.97	0.89	0.91	0.86	0.85	0.9	0.72
Task-Clock Desprotegido	0.74	0.7	0.67	0.66	0.65	0.65	0.6	0.57
Cycles Overhead (%)	38.0	36.19	33.25	38.6	34.2	32.44	46.27	26.98
Cycles Protegido	2903347.8	2692894.8	2534408.9	2597580.4	2470480.2	2395745.4	2547030.1	2012206.8
Cycles Desprotegido	2103880.0	1977339.9	1901930.5	1874210.6	1840896.9	1808912.7	1741298.6	1584653.0
Instructions Overhead (%)	49.84	45.97	39.22	47.18	42.2	36.97	57.71	30.79
Instructions Protegido	1784340.3	1657250.7	1602963.1	1602491.9	1485038.0	1526014.4	1659654.3	1209700.4
Instructions Desprotegido	1190813.9	1135328.7	1151400.3	1088772.4	1044363.4	1114124.1	1052358.5	924893.82

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	dirname	du	echo	expand	expr	factor	false	fnt
Cache-Misses Overhead (%)	12.25	20.88	16.99	15.7	23.03	18.68	15.22	17.62
Cache-Misses Protegido	3156.44	4535.94	2856.61	3536.31	3331.5	3504.04	2840.74	3444.73
Cache-Misses Desprotegido	2811.89	3752.57	2441.85	3056.43	2707.91	2952.53	2465.53	2928.67
Cache-References Overhead (%)	20.28	25.11	22.58	23.27	27.89	25.19	25.28	22.21
Cache-References Protegido	24328.9	30895.75	23442.05	27432.25	25749.76	26107.19	23803.02	26661.88
Cache-References Desprotegido	20227.23	24694.55	19123.37	22253.36	20134.2	20853.77	18999.26	21816.87
Branch-Misses Overhead (%)	25.01	48.56	28.59	32.22	46.39	37.96	26.7	32.35
Branch-Misses Protegido	10519.37	15019.96	10338.5	12064.42	12761.2	12723.79	10262.49	11989.86
Branch-Misses Desprotegido	8414.67	10110.14	8039.78	9124.66	8717.27	9222.62	8100.07	9059.01
Branches Overhead (%)	23.6	58.51	28.38	37.2	55.22	43.58	25.02	32.16
Branches Protegido	200531.41	324835.29	200443.61	244684.94	267917.11	277838.05	195498.46	244185.97
Branches Desprotegido	162238.34	204932.95	156137.1	178339.74	172603.41	193503.83	156379.99	184770.84
Stalled-Cycles-Frontend Overhead (%)	22.25	40.32	25.59	30.74	40.95	37.05	21.13	27.78
Stalled-Cycles-Frontend Protegido	1353087.5	1982800.2	1324654.7	1582798.1	1603809.3	2202555.4	1299253.0	1561831.2
Stalled-Cycles-Frontend Desprotegido	1106778.4	1413023.4	1054751.6	1210653.6	1137893.5	1607116.4	1072630.0	1222258.3
Task-Clock Overhead (%)	20.0	42.86	22.22	30.0	45.76	39.02	16.67	27.87
Task-Clock Protegido	0.66	1.0	0.66	0.78	0.86	1.14	0.63	0.78
Task-Clock Desprotegido	0.55	0.7	0.54	0.6	0.59	0.82	0.54	0.61
Cycles Overhead (%)	23.83	46.83	27.4	33.85	46.72	38.57	23.13	30.38
Cycles Protegido	1890523.9	2861938.0	1862043.3	2239573.2	2324518.0	3221910.5	1825864.6	2217587.3
Cycles Desprotegido	1526677.8	1949136.9	1461524.4	1673212.8	1584287.0	2325098.1	1482856.7	1700847.1
Instructions Overhead (%)	25.57	59.76	29.41	39.06	56.52	38.13	26.11	33.94
Instructions Protegido	1106599.7	1805858.6	1106937.2	1351292.8	1488554.0	2132956.4	1082376.8	1358663.9
Instructions Desprotegido	881250.52	1130364.1	855385.48	971707.03	951003.39	1544155.7	858246.8	1014381.7

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	fold	ginstall	groups	head	hostid	id	join	link
Cache-Misses Overhead (%)	14.36	19.62	12.5	15.32	13.3	13.26	12.5	18.45
Cache-Misses Protegido	3947.89	4546.91	3314.72	2954.84	3139.49	3912.17	3843.25	3154.1
Cache-Misses Desprotegido	3452.28	3801.12	2946.33	2562.2	2771.04	3454.06	3416.34	2662.73
Cache-References Overhead (%)	19.53	24.34	21.95	22.61	20.31	18.76	19.71	21.93
Cache-References Protegido	32300.76	31556.39	26178.4	26620.65	24330.36	29280.44	31510.0	24361.86
Cache-References Desprotegido	27022.14	25378.85	21466.09	21710.94	20222.26	24656.07	26321.21	19980.23
Branch-Misses Overhead (%)	25.24	44.81	25.24	31.14	24.23	23.77	24.66	27.8
Branch-Misses Protegido	13228.29	15098.91	11327.15	11368.73	10539.67	12819.38	13364.48	10721.01
Branch-Misses Desprotegido	10562.66	10426.37	9044.08	8669.26	8483.73	10357.4	10720.85	8388.69
Branches Overhead (%)	24.17	50.38	23.99	33.55	24.25	23.02	24.56	26.3
Branches Protegido	340292.21	315425.62	220619.66	233126.88	204038.61	258800.49	284998.73	205415.36
Branches Desprotegido	274056.41	209757.89	177928.36	174558.27	164218.39	210371.98	228797.06	162636.34
Stalled-Cycles-Frontend Overhead (%)	27.54	39.47	21.13	26.84	21.51	20.66	23.25	21.57
Stalled-Cycles-Frontend Protegido	2036324.0	1965551.6	1445003.3	1493969.4	1379273.6	1697368.6	1793447.4	1360266.3
Stalled-Cycles-Frontend Desprotegido	1596627.9	1409296.1	1192939.1	1177879.3	1135124.1	1406721.4	1455166.1	1118945.9
Task-Clock Overhead (%)	24.39	42.86	19.67	27.87	29.09	22.06	25.35	23.64
Task-Clock Protegido	1.02	1.0	0.73	0.78	0.71	0.83	0.89	0.68
Task-Clock Desprotegido	0.82	0.7	0.61	0.61	0.55	0.68	0.71	0.55
Cycles Overhead (%)	26.97	45.43	23.05	29.96	23.31	22.27	24.63	23.82
Cycles Protegido	2917473.3	2841679.8	2033557.6	2116687.7	1928031.0	2386729.5	2543349.4	1911707.3
Cycles Desprotegido	2297763.5	1954035.9	1652613.4	1628687.1	1563520.5	1951947.1	2040698.7	1543961.8
Instructions Overhead (%)	23.55	56.74	25.59	35.32	25.33	24.17	25.26	27.13
Instructions Protegido	1841245.1	1805757.9	1216738.6	1285009.0	1129235.0	1426092.1	1562166.3	1135597.6
Instructions Desprotegido	1490269.5	1152076.6	96854.95	949580.15	900991.79	1148513.0	1247148.1	893283.07

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	ln	logname	ls	md5sum	mkdir	mkfifo	mknod	mktemp
Cache-Misses Overhead (%)	15.29	13.61	19.57	2.2	12.77	12.95	13.8	9.92
Cache-Misses Protegido	3396.5	3178.01	4147.97	20298.28	4110.03	3182.94	3066.44	4244.29
Cache-Misses Desprotegido	2946.14	2797.18	3468.98	19860.47	3644.74	2817.97	2694.67	3861.08
Cache-References Overhead (%)	21.04	20.15	28.93	8.36	19.64	20.63	21.22	17.48
Cache-References Protegido	26296.4	24388.11	27937.43	69525.58	28396.52	25156.94	24646.14	27966.53
Cache-References Desprotegido	21724.94	20298.48	21669.32	64163.99	23735.02	20854.9	20331.0	23804.75
Branch-Misses Overhead (%)	27.62	26.4	49.96	14.78	28.81	24.8	25.73	21.77
Branch-Misses Protegido	11549.33	10794.62	13436.49	17688.11	12547.75	10845.12	10747.2	12036.94
Branch-Misses Desprotegido	9049.9	8539.87	8959.77	15410.25	9740.96	8689.68	8547.55	9885.19
Branches Overhead (%)	31.4	25.04	61.29	10.2	32.28	27.21	28.38	23.69
Branches Protegido	221583.14	205231.18	294391.34	582085.41	236689.15	208457.48	210699.37	218389.71
Branches Desprotegido	168637.32	164127.77	182523.8	528228.29	178933.02	163869.12	164124.41	176565.73
Stalled-Cycles-Frontend Overhead (%)	24.59	22.5	43.35	10.5	23.81	22.66	26.26	18.63
Stalled-Cycles-Frontend Protegido	1476398.7	1389652.1	1769310.5	3703774.8	1650535.8	1401540.6	1390272.7	1576652.1
Stalled-Cycles-Frontend Desprotegido	1185042.9	1134448.8	1234264.9	3351697.7	1333161.7	1142587.0	1101118.6	1329040.0
Task-Clock Overhead (%)	28.57	28.57	50.85	7.36	25.0	25.0	22.41	19.4
Task-Clock Protegido	0.72	0.72	0.89	1.75	0.8	0.7	0.71	0.8
Task-Clock Desprotegido	0.56	0.56	0.59	1.63	0.64	0.56	0.58	0.67
Cycles Overhead (%)	27.57	24.53	50.26	10.91	27.05	24.92	27.95	20.67
Cycles Protegido	2071944.0	1941872.4	2562939.1	5152080.9	2287875.9	1960173.2	1953184.1	2164154.6
Cycles Desprotegido	1624171.4	1559381.3	1705721.0	4645201.7	1800773.0	1569189.1	1526575.7	1793475.8
Instructions Overhead (%)	32.71	27.49	63.67	10.76	33.56	28.66	30.01	24.62
Instructions Protegido	1227765.4	1138179.9	1635004.6	3136732.3	1313712.4	1151781.0	1164764.6	1210277.7
Instructions Desprotegido	925115.64	892733.89	998976.26	2831940.9	983624.53	895224.47	895929.28	971154.17

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	mv	nl	nohup	nproc	numfmt	od	paste	pathchk
Cache-Misses Overhead (%)	24.67	14.85	13.66	13.7	14.81	15.92	20.92	15.45
Cache-Misses Protegido	3369.53	3329.8	6001.16	2906.36	3243.66	3602.94	3203.15	2873.35
Cache-Misses Desprotegido	2702.84	2899.28	5279.83	2556.24	2825.16	3108.01	2648.9	2488.88
Cache-References Overhead (%)	25.53	22.81	19.92	21.98	25.18	24.59	21.15	21.88
Cache-References Protegido	28440.79	26849.23	43323.24	25135.09	26639.17	28014.98	27530.32	23939.42
Cache-References Desprotegido	22656.29	21862.39	36126.37	20605.57	21281.5	22486.06	22724.1	19641.48
Branch-Misses Overhead (%)	47.16	31.85	58.93	28.72	35.61	36.46	26.45	24.24
Branch-Misses Protegido	14131.58	12017.39	14376.05	11092.17	12253.86	12954.24	11579.39	10254.16
Branch-Misses Desprotegido	9603.04	9114.55	9045.6	8617.44	9036.41	9493.01	9157.49	8253.31
Branches Overhead (%)	63.31	39.49	24.57	29.86	44.7	39.75	28.41	26.52
Branches Protegido	299336.31	253128.54	401419.22	216206.93	258901.47	277157.9	216814.33	204816.44
Branches Desprotegido	183289.06	181467.29	322243.91	166493.95	178917.1	198328.16	168840.89	161881.93
Stalled-Cycles-Frontend Overhead (%)	43.45	31.05	18.11	24.17	33.95	32.75	25.89	24.32
Stalled-Cycles-Frontend Protegido	1763562.8	1575585.9	2460941.6	1396410.5	1566766.0	1692487.9	1493568.3	1329896.0
Stalled-Cycles-Frontend Desprotegido	1229358.6	1202314.2	2083686.1	1124593.4	1169656.5	1274946.1	1186415.8	1069739.5
Task-Clock Overhead (%)	50.0	27.87	16.98	27.59	35.59	34.85	24.59	25.93
Task-Clock Protegido	1.02	0.78	1.24	0.74	0.8	0.89	0.76	0.68
Task-Clock Desprotegido	0.68	0.61	1.06	0.58	0.59	0.66	0.61	0.54
Cycles Overhead (%)	50.73	34.95	20.17	26.86	38.39	36.39	27.85	26.09
Cycles Protegido	2571592.2	2251494.7	3495707.6	1976413.4	2255840.8	2426035.0	2082095.5	1876770.4
Cycles Desprotegido	1706143.8	1668415.5	2909031.1	1557921.3	1630058.7	1778695.9	1628562.6	1488390.0
Instructions Overhead (%)	65.34	42.18	23.24	31.11	46.74	41.59	30.64	28.25
Instructions Protegido	1670468.1	1402298.6	2197640.9	1196163.8	1427590.2	1506926.4	1203721.6	1133582.3
Instructions Desprotegido	1010321.7	986255.65	1783197.9	912313.14	972880.68	1064299.8	921375.68	883909.92

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	pinky	pr	printenv	printf	ptx	pwd	readlink	realpath
Cache-Misses Overhead (%)	14.41	17.29	12.44	16.56	18.09	16.79	16.36	15.18
Cache-Misses Protegido	3213.54	3791.79	2958.21	3500.61	3479.91	3091.52	2927.23	2989.83
Cache-Misses Desprotegido	2808.84	3232.97	2630.85	3003.26	2946.9	2647.07	2515.71	2595.69
Cache-References Overhead (%)	21.48	25.7	21.47	20.5	24.41	20.75	23.99	22.66
Cache-References Protegido	24926.97	29652.05	24123.58	32138.76	26399.36	26211.26	24661.92	24578.75
Cache-References Desprotegido	20519.71	23589.4	19860.46	26671.35	21218.91	21706.6	19890.02	20038.11
Branch-Misses Overhead (%)	27.89	24.71	25.13	29.61	35.3	25.17	34.14	30.17
Branch-Misses Protegido	11072.95	13478.88	10330.53	13772.51	12466.71	11028.55	11273.84	11085.05
Branch-Misses Desprotegido	8658.31	10808.08	8255.52	10625.77	9214.3	8811.18	8404.79	8516.14
Branches Overhead (%)	29.76	40.72	26.62	27.14	46.79	24.7	33.82	32.18
Branches Protegido	222041.55	318028.3	202852.59	312055.35	262052.08	219541.0	220767.8	219473.83
Branches Desprotegido	171119.18	225997.9	160208.91	245443.59	178525.41	176056.19	164978.45	166035.53
Stalled-Cycles-Frontend Overhead (%)	26.94	35.5	20.72	25.34	33.43	22.35	27.95	25.19
Stalled-Cycles-Frontend Protegido	1426994.3	1808472.8	1335847.0	1822375.8	1599470.3	1437475.5	1404987.3	1392509.2
Stalled-Cycles-Frontend Desprotegido	1124178.6	1334715.3	1106549.2	1453892.2	1198739.3	1174932.3	1098048.1	1112287.3
Task-Clock Overhead (%)	29.09	39.39	17.86	26.03	35.59	25.0	32.73	28.57
Task-Clock Protegido	0.71	0.92	0.66	0.92	0.8	0.75	0.73	0.72
Task-Clock Desprotegido	0.55	0.66	0.56	0.73	0.59	0.6	0.55	0.56
Cycles Overhead (%)	29.29	38.01	22.71	27.11	38.39	23.92	30.86	28.38
Cycles Protegido	2020225.5	2606328.0	1878632.8	2630769.4	2300271.2	2023025.8	1998065.9	1979549.6
Cycles Desprotegido	1562519.9	1888508.2	1530891.1	2069664.2	1662140.4	1632482.0	1526849.8	1541989.3
Instructions Overhead (%)	32.03	39.78	25.38	28.3	47.74	25.51	35.05	33.64
Instructions Protegido	1225457.7	1660151.0	1121304.3	1703297.7	1450758.5	1209231.7	1222981.4	1212948.9
Instructions Desprotegido	928145.75	1187678.1	894310.16	1327607.5	981975.81	963476.27	905562.79	907593.97

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	rm	rmdir	seq	sha1sum	sha224sum	sha256sum	sha384sum	sha512sum
Cache-Misses Overhead (%)	14.22	13.23	15.09	1.81	0.9	2.1	1.37	2.83
Cache-Misses Protegido	4711.83	3195.93	3227.58	19380.18	19378.61	19298.98	19305.37	19612.09
Cache-Misses Desprotegido	4125.14	2822.41	2804.29	19034.97	19206.55	18901.9	19044.66	19072.73
Cache-References Overhead (%)	21.51	21.66	23.02	7.75	8.01	7.23	7.89	8.1
Cache-References Protegido	31765.6	25143.75	25209.52	64090.14	62991.68	62665.72	62913.25	63004.18
Cache-References Desprotegido	26143.12	20667.56	20492.86	59478.2	58318.81	58440.78	58313.07	58280.76
Branch-Misses Overhead (%)	29.91	28.0	35.55	15.73	17.02	16.62	16.71	16.99
Branch-Misses Protegido	13708.11	11106.37	11733.85	17052.34	17164.66	17107.22	17107.98	17177.79
Branch-Misses Desprotegido	10552.07	8676.99	8656.74	14734.54	14667.55	14669.33	14659.16	14682.72
Branches Overhead (%)	37.36	28.21	34.91	10.09	10.63	10.47	10.5	10.81
Branches Protegido	281759.26	212387.08	251167.01	550068.27	556058.11	555323.93	555323.9	556906.34
Branches Desprotegido	205129.19	165651.05	186176.28	499654.85	502641.86	502681.4	502575.12	502598.35
Stalled-Cycles-Frontend Overhead (%)	27.42	24.59	29.68	8.33	8.75	10.72	11.79	9.54
Stalled-Cycles-Frontend Protegido	1904528.4	1418366.5	1508484.1	3453997.2	3459407.1	3454633.0	3475083.1	3472613.3
Stalled-Cycles-Frontend Desprotegido	1494690.4	1138462.4	1163197.1	3188471.3	3181131.2	3120196.2	3108692.5	3170106.2
Task-Clock Overhead (%)	30.56	24.14	33.9	10.07	12.5	10.69	9.62	8.23
Task-Clock Protegido	0.94	0.72	0.79	1.64	1.71	1.76	1.71	1.71
Task-Clock Desprotegido	0.72	0.58	0.59	1.49	1.52	1.59	1.56	1.58
Cycles Overhead (%)	31.11	26.6	32.35	9.37	9.66	11.17	11.87	10.33
Cycles Protegido	2666756.1	1986632.5	2179097.1	4831973.7	4933027.2	4926965.9	4925007.9	4928830.8
Cycles Desprotegido	2033966.2	1569250.3	1646469.6	4418083.2	4498391.8	4431791.4	4402531.8	4467495.2
Instructions Overhead (%)	38.52	29.25	35.74	10.6	10.8	11.1	10.73	10.9
Instructions Protegido	1570624.8	1172417.6	1375205.6	3015361.4	3187469.2	3184509.1	3145115.5	3160481.1
Instructions Desprotegido	1133846.5	907092.72	1013116.2	2726478.0	2876791.6	2866469.6	2840427.0	2849951.0

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	shred	shuf	sleep	sort	split	stat	stdbuf	sum
Cache-Misses Overhead (%)	14.62	15.95	26.47	2.63	13.15	20.27	15.31	15.71
Cache-Misses Protegido	3986.83	3119.16	2560.44	46063.02	5260.06	3208.0	2966.54	3712.97
Cache-Misses Desprotegido	3478.43	2690.11	2024.6	44880.51	4648.94	2667.37	2572.73	3208.93
Cache-References Overhead (%)	19.7	24.49	23.84	7.51	18.45	26.13	22.61	20.29
Cache-References Protegido	34610.98	25698.56	24846.11	119676.06	37821.39	25955.27	25090.96	29074.97
Cache-References Desprotegido	28914.05	20642.7	20063.68	111313.01	31930.93	20578.12	20463.56	24171.0
Branch-Misses Overhead (%)	26.04	31.78	30.38	20.8	28.78	41.69	29.22	28.4
Branch-Misses Protegido	15004.38	11625.48	11142.24	23637.49	14450.55	12171.47	11184.35	11728.59
Branch-Misses Desprotegido	11904.67	8822.07	8546.05	19566.85	11220.87	8590.0	8655.54	9134.42
Branches Overhead (%)	33.89	34.32	32.36	13.76	34.28	49.47	33.16	31.49
Branches Protegido	272013.86	257829.62	212466.4	1214164.6	315444.42	256754.18	223617.34	278545.31
Branches Desprotegido	203165.42	191947.41	160527.46	1067336.4	234909.53	171772.31	167932.56	211831.93
Stalled-Cycles-Frontend Overhead (%)	25.63	30.43	28.93	11.1	25.31	35.91	28.86	24.69
Stalled-Cycles-Frontend Protegido	1867564.8	1585881.3	1367064.1	8423836.6	2150722.0	1545824.1	1434553.3	1773983.9
Stalled-Cycles-Frontend Desprotegido	1486583.5	1215894.1	1060310.7	7582191.8	1716262.0	1137400.1	1113276.4	1422709.3
Task-Clock Overhead (%)	28.4	32.26	27.87	12.98	28.75	41.07	33.93	33.33
Task-Clock Protegido	1.04	0.82	0.78	4.09	1.03	0.79	0.75	0.92
Task-Clock Desprotegido	0.81	0.62	0.61	3.62	0.8	0.56	0.56	0.69
Cycles Overhead (%)	28.58	32.45	31.23	12.7	28.6	41.34	31.76	27.8
Cycles Protegido	2613746.0	2284331.7	1935879.9	11602627.	3014144.7	2235478.0	2034482.2	2529748.3
Cycles Desprotegido	2032711.4	1724683.1	1475155.7	10295246.	2343809.7	1581635.0	1544118.3	1979393.8
Instructions Overhead (%)	33.51	34.46	33.76	15.34	35.32	51.17	35.77	33.31
Instructions Protegido	1525950.9	1445800.6	1172981.1	6992504.8	1783756.9	1427760.3	1238555.1	1579839.8
Instructions Desprotegido	1142989.5	1075297.6	876898.7	6062718.4	1318180.7	944465.4	912229.25	1185066.8

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	sync	tac	tail	tee	test	touch	tr	true
Cache-Misses Overhead (%)	16.5	13.15	22.44	9.38	19.46	18.23	17.74	17.45
Cache-Misses Protegido	3062.98	4275.87	3636.06	3417.53	2836.34	3253.75	3241.77	2408.83
Cache-Misses Desprotegido	2629.18	3779.01	2969.67	3124.53	2374.39	2751.96	2753.34	2051.0
Cache-References Overhead (%)	21.34	21.02	26.36	19.04	26.02	21.89	24.17	24.68
Cache-References Protegido	24669.22	30680.53	27649.6	28107.7	23247.59	28620.21	25594.96	21727.38
Cache-References Desprotegido	20330.28	25352.46	21882.27	23612.24	18448.18	23479.51	20613.66	17427.07
Branch-Misses Overhead (%)	24.03	34.62	42.01	26.72	39.52	36.78	31.97	28.54
Branch-Misses Protegido	10660.16	13501.95	12754.88	11924.58	10807.7	13075.22	11135.89	9566.37
Branch-Misses Desprotegido	8595.0	10029.74	8981.41	9409.81	7746.26	9559.48	8437.95	7442.34
Branches Overhead (%)	25.34	41.28	49.66	26.54	44.89	45.4	36.85	26.78
Branches Protegido	205729.89	277948.01	258740.84	237665.45	224920.71	264149.83	228549.47	183394.03
Branches Desprotegido	164135.51	196730.38	172888.94	187812.88	155240.68	181674.52	167001.29	144652.11
Stalled-Cycles-Frontend Overhead (%)	22.58	29.93	36.76	23.87	34.38	42.83	32.57	25.87
Stalled-Cycles-Frontend Protegido	1381515.0	1820483.6	1644698.4	1592233.9	1367411.6	1636153.0	1494058.2	1187157.6
Stalled-Cycles-Frontend Desprotegido	1127017.1	1401073.6	1202573.3	1285393.7	1017578.3	1145539.4	1126967.0	943153.44
Task-Clock Overhead (%)	22.41	31.08	41.67	25.76	36.73	47.27	33.33	18.18
Task-Clock Protegido	0.71	0.97	0.85	0.83	0.67	0.81	0.72	0.65
Task-Clock Desprotegido	0.58	0.74	0.6	0.66	0.49	0.55	0.54	0.55
Cycles Overhead (%)	24.17	34.36	41.73	26.18	38.8	46.56	35.15	27.03
Cycles Protegido	1932183.1	2567111.1	2342968.8	2234683.5	1969276.7	2343332.3	2119091.4	1679320.4
Cycles Desprotegido	1556045.0	1910668.9	1653074.6	1771091.1	1418752.7	1598907.5	1568005.8	1321949.4
Instructions Overhead (%)	26.11	43.54	51.34	29.66	46.71	53.18	38.9	27.76
Instructions Protegido	1135913.4	1542131.9	1432286.7	1321832.3	1244843.0	1452873.6	1275128.3	1015070.7
Instructions Desprotegido	900726.78	1074345.6	946372.85	1019490.3	848516.97	948479.06	917989.01	794502.23

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O0	-O0
Programa	truncate	tsort	tty	uname	unexpand	uniq	unlink	users
Cache-Misses Overhead (%)	14.28	15.79	16.13	15.33	13.58	15.56	14.48	15.73
Cache-Misses Protegido	3135.15	3319.85	2913.69	2833.56	3519.16	3472.45	3158.63	3169.51
Cache-Misses Desprotegido	2743.28	2867.13	2509.05	2456.84	3098.51	3004.89	2759.09	2738.62
Cache-References Overhead (%)	21.07	20.69	22.69	21.88	21.65	19.01	21.2	21.56
Cache-References Protegido	25430.48	25594.59	24169.46	23193.63	27265.44	30831.85	24213.86	24501.35
Cache-References Desprotegido	21004.32	21206.4	19699.08	19029.62	22413.75	25906.2	19978.39	20155.99
Branch-Misses Overhead (%)	24.02	29.07	28.47	27.86	31.24	23.95	26.76	25.94
Branch-Misses Protegido	10954.39	11232.7	10621.62	10104.7	12050.46	12733.83	10680.66	10733.0
Branch-Misses Desprotegido	8832.9	8702.47	8268.05	7902.95	9182.1	10273.66	8425.84	8522.16
Branches Overhead (%)	26.99	27.35	28.95	26.25	36.27	22.7	25.77	28.03
Branches Protegido	211628.51	215611.85	208388.36	197274.33	243116.13	268665.53	204311.53	211194.6
Branches Desprotegido	166646.96	169307.57	161605.42	156253.73	178404.89	218953.43	162444.33	164950.8
Stalled-Cycles-Frontend Overhead (%)	22.85	22.82	24.4	23.74	28.2	21.57	22.52	23.37
Stalled-Cycles-Frontend Protegido	1421332.6	1447581.7	1354544.1	1295296.0	1571132.9	1693129.3	1355476.9	1392449.4
Stalled-Cycles-Frontend Desprotegido	1156995.5	1178633.4	1088863.6	1046772.3	1225566.5	1392670.3	1106359.8	1128706.0
Task-Clock Overhead (%)	22.03	23.73	27.27	25.0	33.9	23.19	27.78	25.0
Task-Clock Protegido	0.72	0.73	0.7	0.65	0.79	0.85	0.69	0.7
Task-Clock Desprotegido	0.59	0.59	0.55	0.52	0.59	0.69	0.54	0.56
Cycles Overhead (%)	24.91	25.13	26.88	25.55	31.67	22.94	24.43	25.86
Cycles Protegido	1986428.2	2028167.3	1913410.4	1823042.4	2225426.4	2400838.3	1901955.1	1959305.4
Cycles Desprotegido	1590226.7	1620808.6	1508014.0	1452022.5	1690190.6	1952896.7	1528589.9	1556691.7
Instructions Overhead (%)	28.18	28.77	30.64	27.49	38.18	23.81	26.64	29.6
Instructions Protegido	1166928.2	1195577.0	1152512.5	1089551.8	1344488.1	1471412.9	1125833.5	1170230.2
Instructions Desprotegido	910413.91	928438.23	882189.9	854596.66	973012.45	1188478.0	888974.71	902960.17

Otimização	-O0	-O0	-O0	-O0	-O0	-O0	-O2	-O2	-O2
Programa	vdtr	wc	who	whoami	yes	l	b2sum	base32	
Cache-Misses Overhead (%)	19.97	6.2	17.29	15.11	7.75	18.81	19.58	18.62	
Cache-Misses Protegido	4046.85	8107.51	3228.8	3235.94	4413.98	2971.84	3575.91	3765.57	
Cache-Misses Desprotegido	3373.29	7634.09	2752.81	2811.14	4096.68	2501.29	2990.49	3174.58	
Cache-References Overhead (%)	26.28	17.8	23.3	21.6	17.3	21.5	25.71	21.11	
Cache-References Protegido	28147.0	33609.59	25088.82	24766.88	29797.99	23865.79	27242.63	26527.08	
Cache-References Desprotegido	22289.88	28531.55	20347.67	20367.25	25403.65	19643.07	21671.48	21902.68	
Branch-Misses Overhead (%)	45.97	31.58	35.62	27.22	27.91	39.8	41.58	34.85	
Branch-Misses Protegido	14147.84	12555.69	11824.2	10872.45	11811.26	11567.08	12784.47	12075.26	
Branch-Misses Desprotegido	9692.47	9542.44	8718.83	8546.19	9234.21	8274.06	9030.08	8954.67	
Branches Overhead (%)	55.57	23.37	38.6	26.17	23.13	52.38	53.27	44.1	
Branches Protegido	301538.47	344764.35	238953.26	208392.14	260520.81	248780.7	276981.22	242473.92	
Branches Desprotegido	193822.14	279461.07	172404.18	165161.22	211587.28	163259.93	180718.13	168263.48	
Stalled-Cycles-Frontend Overhead (%)	39.52	19.51	29.21	20.53	21.54	34.23	37.65	30.36	
Stalled-Cycles-Frontend Protegido	1754656.7	2123782.7	1471839.2	1374493.5	1704921.7	1472006.8	1657359.2	1560909.7	
Stalled-Cycles-Frontend Desprotegido	1257619.7	1777119.9	1139069.3	1140366.9	1402737.9	1096640.6	1204024.4	1197386.2	
Task-Clock Overhead (%)	45.16	20.88	32.14	33.33	19.44	36.84	40.98	35.09	
Task-Clock Protegido	0.9	1.1	0.74	0.72	0.86	0.78	0.86	0.77	
Task-Clock Desprotegido	0.62	0.91	0.56	0.54	0.72	0.57	0.61	0.57	
Cycles Overhead (%)	45.95	21.24	33.27	23.14	23.13	40.83	43.43	35.26	
Cycles Protegido	2557030.9	3028542.4	2109643.8	1933374.6	2406945.0	2138476.2	2401904.3	2215637.8	
Cycles Desprotegido	1751967.2	2497974.2	1582931.8	1570063.6	1954846.4	1518437.5	1674594.9	1638087.6	
Instructions Overhead (%)	57.53	23.58	40.13	27.25	24.95	54.71	55.41	45.84	
Instructions Protegido	1667094.8	1899834.0	1320941.5	1149736.5	1448583.3	1375595.1	1534517.2	1342898.8	
Instructions Desprotegido	1058266.8	1537303.8	942640.7	903548.62	1159341.8	889116.7	987380.26	920804.61	

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	base64	basename	basenc	cat	chgrp	chmod	chown	chroot
Cache-Misses Overhead (%)	17.72	22.76	31.25	17.75	20.88	29.87	24.21	17.56
Cache-Misses Protegido	3220.51	3109.43	4226.24	4322.8	5001.79	4278.46	4251.45	3143.33
Cache-Misses Desprotegido	2735.85	2532.91	3219.9	3671.21	4137.88	3294.33	3422.76	2673.82
Cache-References Overhead (%)	24.52	23.71	32.29	22.01	22.81	30.06	25.58	27.51
Cache-References Protegido	25052.39	23963.58	29532.31	27687.1	35183.34	30608.15	31351.9	25352.93
Cache-References Desprotegido	20118.98	19371.44	22324.7	22693.04	28648.64	23534.51	24964.81	19883.56
Branch-Misses Overhead (%)	40.84	39.2	86.96	39.94	44.38	54.01	50.7	40.53
Branch-Misses Protegido	11999.24	11219.81	16882.21	12029.64	17366.29	14649.34	16085.54	11892.58
Branch-Misses Desprotegido	8519.54	8060.14	9029.64	8596.29	12027.88	9511.73	10673.92	8462.89
Branches Overhead (%)	49.27	44.61	123.87	45.64	61.17	77.33	67.77	47.91
Branches Protegido	245983.34	227694.43	400630.24	263675.78	388185.94	329259.19	357163.92	245657.64
Branches Desprotegido	164786.31	157459.28	178958.03	181049.48	240855.35	185680.68	212883.63	166091.11
Stalled-Cycles-Frontend Overhead (%)	34.47	33.23	71.41	31.08	41.07	51.25	44.51	38.02
Stalled-Cycles-Frontend Protegido	1501805.7	1420834.5	2124086.9	1688302.5	2243968.7	1971989.1	2035166.1	1522394.8
Stalled-Cycles-Frontend Desprotegido	1116815.1	1066439.1	1239213.2	1287997.8	1590718.7	1303752.3	1408317.0	1103063.2
Task-Clock Overhead (%)	38.18	33.96	83.33	34.92	49.37	60.94	50.0	35.09
Task-Clock Protegido	0.76	0.71	1.1	0.85	1.18	1.03	1.05	0.77
Task-Clock Desprotegido	0.55	0.53	0.6	0.63	0.79	0.64	0.7	0.57
Cycles Overhead (%)	40.07	37.86	88.2	36.12	48.47	60.12	52.91	42.54
Cycles Protegido	2165981.9	2033955.0	3206479.1	2403929.1	3281045.6	2871019.9	2994844.9	2183694.4
Cycles Desprotegido	1546382.1	1475360.2	1703755.8	1766015.4	2209871.4	1793011.0	1958505.8	1531948.4
Instructions Overhead (%)	51.44	46.27	128.25	47.25	63.82	79.88	70.41	50.88
Instructions Protegido	1367624.7	1259867.1	2234631.9	1461762.7	2151443.1	1836398.8	1984521.1	1366522.5
Instructions Desprotegido	903093.48	861331.5	979030.88	992689.23	1313328.3	1020902.0	1164536.8	905723.65

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	cksum	comm	cp	csplit	cut	date	dd	df	
Cache-Misses Overhead (%)	4.24	20.32	25.57	31.97	25.7	21.9	23.75	30.52	
Cache-Misses Protegido	21281.69	3521.14	5508.37	4449.11	3727.79	4021.49	4360.9	4091.48	
Cache-Misses Desprotegido	20416.17	2926.57	4386.6	3371.41	2965.54	3298.89	3523.91	3134.66	
Cache-References Overhead (%)	11.64	23.85	28.81	28.96	23.57	27.56	28.15	29.17	
Cache-References Protegido	71436.06	26699.04	34466.98	34043.25	31001.08	31208.12	30486.71	29057.67	
Cache-References Desprotegido	63987.49	21557.46	26759.02	26397.58	25086.88	24465.99	23790.51	22495.04	
Branch-Misses Overhead (%)	34.93	41.17	71.64	65.18	42.84	48.35	48.13	58.27	
Branch-Misses Protegido	23465.66	12560.95	18467.19	17178.63	14301.98	15150.36	14228.77	15757.89	
Branch-Misses Desprotegido	17391.3	8897.64	10758.98	10400.01	10012.66	10212.31	9605.28	9956.6	
Branches Overhead (%)	31.09	51.14	104.22	100.42	55.52	65.73	68.87	80.48	
Branches Protegido	791492.13	259365.81	438577.52	412811.12	328373.92	327211.4	318377.41	374362.38	
Branches Desprotegido	603776.25	171604.06	214758.88	205969.73	211147.02	197437.86	188537.22	207431.22	
Stalled-Cycles-Frontend Overhead (%)	21.95	38.0	62.25	57.25	39.12	45.13	45.33	56.01	
Stalled-Cycles-Frontend Protegido	4330124.7	1627242.0	2455131.7	2275343.8	1880055.7	1932900.5	1930120.6	1976944.4	
Stalled-Cycles-Frontend Desprotegido	3550749.4	1179136.0	1513209.5	1446925.6	1351365.1	1331838.3	1328112.1	1267220.0	
Task-Clock Overhead (%)	27.53	39.66	72.97	67.61	43.28	50.77	51.56	63.08	
Task-Clock Protegido	2.27	0.81	1.28	1.19	0.96	0.98	0.97	1.06	
Task-Clock Desprotegido	1.78	0.58	0.74	0.71	0.67	0.65	0.64	0.65	
Cycles Overhead (%)	25.68	43.36	75.32	70.5	45.38	52.78	53.27	66.09	
Cycles Protegido	6349983.0	2329813.8	3640487.5	3389381.1	2755722.6	2816013.9	2795013.2	2947827.9	
Cycles Desprotegido	5052595.1	1625122.7	2076450.6	1987878.2	1895479.7	1843162.9	1823530.8	1774875.8	
Instructions Overhead (%)	32.88	54.21	107.28	102.6	57.75	68.81	71.21	85.72	
Instructions Protegido	4327853.0	1442368.6	2454408.1	2303061.0	1815641.4	1815782.9	1775276.2	2026800.6	
Instructions Desprotegido	3256863.5	935303.73	1184086.0	1136743.6	1150962.7	1075658.4	1036917.0	1091301.5	

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	dir	dircolors	dirname	du	echo	expand	expr	factor
Cache-Misses Overhead (%)	35.23	23.59	22.07	40.21	22.85	18.08	68.11	35.8
Cache-Misses Protegido	4382.5	3378.95	3234.35	5086.03	2978.18	3518.5	4352.27	4039.05
Cache-Misses Desprotegido	3240.89	2733.9	2649.67	3627.51	2424.23	2979.79	2588.9	2974.37
Cache-References Overhead (%)	31.97	26.46	23.79	36.39	22.75	22.89	47.23	35.12
Cache-References Protegido	29268.58	26019.8	24678.6	33136.35	23666.58	27202.78	29422.32	28117.17
Cache-References Desprotegido	22177.46	20574.73	19935.99	24295.26	19279.82	22136.69	19983.64	20809.6
Branch-Misses Overhead (%)	79.16	38.95	38.28	94.38	35.77	35.27	121.77	89.02
Branch-Misses Protegido	17362.74	11996.76	11512.64	19443.46	10935.93	12323.22	19328.02	16821.75
Branch-Misses Desprotegido	9691.37	8634.0	8325.68	10002.78	8054.96	9109.85	8715.28	8899.63
Branches Overhead (%)	122.53	51.01	45.36	147.92	44.01	43.04	188.05	122.74
Branches Protegido	427809.2	254154.22	234110.11	500462.72	224886.32	254371.27	490145.97	410140.59
Branches Desprotegido	192249.69	168303.97	161053.79	201866.09	156158.08	177835.03	170157.38	184138.35
Stalled-Cycles-Frontend Overhead (%)	76.1	36.65	35.47	87.87	33.38	31.55	116.41	77.05
Stalled-Cycles-Frontend Protegido	2175496.0	1555967.1	1468782.7	2595903.0	1398781.3	1594856.9	2401457.0	2504547.5
Stalled-Cycles-Frontend Desprotegido	1235363.2	1138615.1	1084223.6	1381756.0	1048727.9	1212386.7	1109704.1	1414569.3
Task-Clock Overhead (%)	88.52	42.86	35.19	104.41	35.85	30.0	134.48	87.32
Task-Clock Protegido	1.15	0.8	0.73	1.39	0.72	0.78	1.36	1.33
Task-Clock Desprotegido	0.61	0.56	0.54	0.68	0.53	0.6	0.58	0.71
Cycles Overhead (%)	91.96	41.81	39.7	106.9	37.76	36.1	140.72	88.86
Cycles Protegido	3315796.2	2235675.0	2099178.2	3950281.4	2003880.8	2278573.5	3727538.7	3890690.0
Cycles Desprotegido	1727328.5	1576521.3	1502646.1	1909290.3	1454612.2	1674218.1	1548481.2	2060054.0
Instructions Overhead (%)	127.3	52.63	47.74	152.35	46.36	45.24	194.9	115.84
Instructions Protegido	2382702.9	1403410.4	1295601.7	2802956.0	1247360.9	1409578.9	2735998.7	2889308.5
Instructions Desprotegido	1048264.6	919456.09	876971.51	1110745.0	852232.01	970503.94	927781.07	1338650.1

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	false	fmt	fold	ginstall	groups	head	hostid	id
Cache-Misses Overhead (%)	23.9	22.77	16.82	32.98	18.2	19.08	20.18	16.99
Cache-Misses Protegido	2919.01	3502.78	3962.57	4872.64	3354.36	2972.71	3212.77	3933.39
Cache-Misses Desprotegido	2355.98	2853.23	3392.06	3664.16	2837.86	2496.47	2673.33	3362.25
Cache-References Overhead (%)	22.71	25.73	20.4	31.52	25.02	24.39	24.53	20.96
Cache-References Protegido	23989.14	27321.49	32394.24	33113.32	26591.19	26884.68	25108.12	29700.66
Cache-References Desprotegido	19549.64	21730.33	26904.61	25176.64	21268.91	21613.61	20162.96	24553.12
Branch-Misses Overhead (%)	33.53	44.01	49.38	78.19	37.71	43.71	37.13	31.16
Branch-Misses Protegido	10869.82	13044.42	15847.49	18495.34	12425.81	12653.27	11664.02	13530.14
Branch-Misses Desprotegido	8140.32	9057.85	10609.12	10379.29	9023.27	8804.67	8505.65	10316.0
Branches Overhead (%)	39.48	52.31	36.85	115.27	44.45	53.15	44.42	37.46
Branches Protegido	218068.01	279149.77	366429.55	435086.66	254850.36	266074.25	236945.19	287574.85
Branches Desprotegido	156345.9	183276.09	267755.94	202109.23	176426.04	173730.84	164064.07	209213.51
Stalled-Cycles-Frontend Overhead (%)	29.53	35.99	36.99	70.32	34.2	36.07	33.81	28.49
Stalled-Cycles-Frontend Protegido	1354565.5	1662072.1	2009265.9	2383595.6	1592113.5	1597000.2	1485720.0	1793142.8
Stalled-Cycles-Frontend Desprotegido	1045716.5	1222163.5	1466725.7	1399472.0	1186345.4	1173685.8	1110344.5	1395513.8
Task-Clock Overhead (%)	42.31	40.98	35.9	85.51	36.67	41.67	38.18	32.35
Task-Clock Protegido	0.74	0.86	1.06	1.28	0.82	0.85	0.76	0.9
Task-Clock Desprotegido	0.52	0.61	0.78	0.69	0.6	0.6	0.55	0.68
Cycles Overhead (%)	33.54	41.95	35.56	84.78	38.35	42.34	38.23	32.16
Cycles Protegido	1943439.9	2408817.9	2960497.8	3590733.1	2276180.9	2314228.1	2124875.7	2561270.3
Cycles Desprotegido	1455300.1	1696977.3	2183916.8	1943202.4	1645280.8	1625873.6	1537235.2	1938028.3
Instructions Overhead (%)	41.03	53.77	38.37	117.51	45.98	55.5	46.52	39.14
Instructions Protegido	1208275.8	1545641.5	1984429.8	2497330.2	1409567.8	1466934.5	1314425.6	1589008.6
Instructions Desprotegido	856775.71	1005166.2	1434132.5	1148146.5	965580.73	943370.68	897085.08	1142035.3

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	join	link	ln	logname	ls	md5sum	mkdir	mkfifo
Cache-Misses Overhead (%)	20.82	21.01	27.2	20.9	46.56	0.12	16.58	19.51
Cache-Misses Protegido	4058.55	3222.79	3671.07	3225.68	5014.13	19940.5	4156.7	3255.56
Cache-Misses Desprotegido	3359.18	2663.14	2885.95	2667.97	3421.11	19916.37	3565.43	2724.01
Cache-References Overhead (%)	23.68	24.2	26.94	23.96	40.22	7.85	22.68	23.44
Cache-References Protegido	32707.88	25002.6	27411.64	25140.82	30232.53	69282.11	29000.6	25178.37
Cache-References Desprotegido	26446.43	20130.36	21593.88	20281.9	21561.27	64242.25	23639.88	20397.64
Branch-Misses Overhead (%)	40.4	35.0	56.67	36.62	88.02	19.84	37.08	35.25
Branch-Misses Protegido	15151.99	11470.79	14246.15	11587.86	16862.91	18370.65	13384.96	11597.5
Branch-Misses Desprotegido	10791.81	8496.96	9092.95	8482.01	8968.45	15329.56	9764.55	8574.71
Branches Overhead (%)	52.34	44.22	86.27	44.29	138.2	15.05	51.86	45.04
Branches Protegido	346551.59	236631.75	313895.57	236002.2	432589.71	606619.39	271045.78	236225.07
Branches Desprotegido	227493.05	164082.61	168512.45	163565.28	181607.85	527260.83	178484.32	162867.28
Stalled-Cycles-Frontend Overhead (%)	37.2	32.7	53.61	32.32	85.85	13.43	33.66	30.37
Stalled-Cycles-Frontend Protegido	1995466.3	1486624.0	1806150.4	1480073.1	2274582.2	3762084.4	1775047.7	1486133.0
Stalled-Cycles-Frontend Desprotegido	1454431.3	1120322.7	1175836.4	1118550.9	1223869.5	3316619.7	1327999.2	1139923.1
Task-Clock Overhead (%)	40.85	32.73	59.65	33.93	101.69	12.35	37.5	32.76
Task-Clock Protegido	1.0	0.73	0.91	0.75	1.19	1.82	0.88	0.77
Task-Clock Desprotegido	0.71	0.55	0.57	0.56	0.59	1.62	0.64	0.58
Cycles Overhead (%)	43.01	37.28	64.16	37.17	102.9	14.61	39.67	35.47
Cycles Protegido	2914177.5	2124270.6	2653084.0	2117518.1	3440855.9	5279068.0	2507428.8	2120902.3
Cycles Desprotegido	2037679.0	1547431.9	1616119.7	1543758.6	1695835.7	4605967.7	1795292.5	1565636.7
Instructions Overhead (%)	54.33	46.35	89.34	46.72	143.06	15.92	53.71	46.38
Instructions Protegido	1910380.4	1313264.5	1748825.3	1310788.5	2416406.1	3271766.4	1507574.0	1307193.4
Instructions Desprotegido	1237843.9	897318.02	923656.99	893415.75	994167.07	2822415.0	980812.58	893012.13

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	mknod	mktemp	mv	nl	nohup	nproc	numfmt	od
Cache-Misses Overhead (%)	22.7	15.55	39.67	28.45	16.76	18.35	24.23	23.61
Cache-Misses Protegido	3186.71	4424.81	3696.68	3678.62	6040.55	2945.49	3401.42	3677.36
Cache-Misses Desprotegido	2597.13	3829.45	2646.81	2863.84	5173.42	2488.7	2737.93	2975.03
Cache-References Overhead (%)	25.49	22.31	33.67	29.5	20.87	23.89	28.3	28.26
Cache-References Protegido	25400.73	29098.19	30036.48	28314.08	43600.61	25284.03	27041.36	28543.83
Cache-References Desprotegido	20241.57	23790.5	22469.87	21864.67	36071.77	20408.19	21076.91	22254.77
Branch-Misses Overhead (%)	41.95	32.03	80.48	66.39	23.29	37.14	50.51	50.75
Branch-Misses Protegido	11965.34	13077.4	17301.97	15210.03	12515.21	11778.54	13395.88	13978.27
Branch-Misses Desprotegido	8429.34	9904.65	9586.63	9141.03	10150.71	8588.42	8900.61	9272.73
Branches Overhead (%)	48.81	46.87	124.83	97.77	38.69	45.11	67.39	64.34
Branches Protegido	243986.97	259369.81	411464.96	359874.03	447832.76	241799.06	296955.41	307361.83
Branches Desprotegido	163960.37	176598.29	183013.5	181965.63	322899.48	166629.18	177400.52	187032.43
Stalled-Cycles-Frontend Overhead (%)	35.45	30.91	75.63	58.11	28.65	31.64	44.87	45.2
Stalled-Cycles-Frontend Protegido	1511036.1	1731957.7	2161459.2	1926462.2	2598236.4	1477040.9	1705263.3	1788654.0
Stalled-Cycles-Frontend Desprotegido	1115569.8	1322988.4	1230718.7	1218424.2	2019619.7	1121989.3	1177084.4	1231845.3
Task-Clock Overhead (%)	40.0	34.85	91.18	68.85	36.63	36.21	50.0	50.0
Task-Clock Protegido	0.77	0.89	1.3	1.03	1.38	0.79	0.9	0.96
Task-Clock Desprotegido	0.55	0.66	0.68	0.61	1.01	0.58	0.6	0.64
Cycles Overhead (%)	40.68	36.26	91.78	71.27	33.01	36.81	52.84	52.31
Cycles Protegido	2168549.1	2435043.0	3274665.9	2890791.1	3763349.2	2127451.5	2501879.8	2616621.5
Cycles Desprotegido	1541466.8	1787078.4	1707526.3	1687849.8	2829374.9	1555082.0	1636901.9	1718007.3
Instructions Overhead (%)	50.88	48.71	128.11	101.17	41.09	47.01	69.58	66.6
Instructions Protegido	1351279.0	1441974.8	2302927.6	1997523.8	2469523.3	1340513.6	1647539.1	1704519.3
Instructions Desprotegido	895599.17	969642.9	1009554.7	992929.04	1750267.3	911859.36	971522.41	1023151.2

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	paste	pathchk	pinky	pr	printenv	printf	ptx	pwd
Cache-Misses Overhead (%)	24.35	25.03	17.49	32.72	19.6	22.04	34.19	21.53
Cache-Misses Protegido	3317.73	3004.86	3161.85	4220.62	3096.74	3637.49	3909.49	3195.47
Cache-Misses Desprotegido	2668.02	2403.38	2691.14	3180.13	2589.24	2980.6	2913.49	2629.31
Cache-References Overhead (%)	23.96	25.56	24.11	31.15	25.69	22.79	33.35	23.63
Cache-References Protegido	28041.21	24197.41	25184.5	30814.63	24726.29	32690.92	28135.41	26632.33
Cache-References Desprotegido	22620.97	19271.49	20292.35	23496.6	19672.86	26623.87	21099.6	21542.61
Branch-Misses Overhead (%)	34.74	36.17	38.79	53.41	38.0	42.05	71.61	35.33
Branch-Misses Protegido	12257.43	11090.37	12078.52	15328.33	11369.29	15029.91	15673.19	11942.88
Branch-Misses Desprotegido	9096.88	8144.46	8702.79	9991.57	8238.8	10581.0	9132.8	8825.33
Branches Overhead (%)	44.14	42.04	50.94	71.27	43.87	46.52	108.99	42.26
Branches Protegido	242633.5	229430.27	254394.57	359266.1	230178.59	357413.75	371935.77	250274.42
Branches Desprotegido	168328.37	161529.08	168539.22	209769.92	159989.39	243928.64	177964.76	175921.37
Stalled-Cycles-Frontend Overhead (%)	32.49	32.22	36.54	51.8	33.31	35.48	71.18	30.79
Stalled-Cycles-Frontend Protegido	1575115.2	1413710.0	1550678.6	1952688.6	1433187.7	1962131.3	2019267.6	1543550.3
Stalled-Cycles-Frontend Desprotegido	1188842.3	1069248.7	1135678.1	1286328.6	1075115.6	1448237.2	1179586.5	1180133.2
Task-Clock Overhead (%)	35.0	32.73	40.0	58.73	32.73	39.73	81.03	36.67
Task-Clock Protegido	0.81	0.73	0.77	1.0	0.73	1.02	1.05	0.82
Task-Clock Desprotegido	0.6	0.55	0.55	0.63	0.55	0.73	0.58	0.6
Cycles Overhead (%)	36.73	36.46	42.09	60.08	37.67	40.45	84.27	35.27
Cycles Protegido	2229751.2	2029410.2	2236468.1	2879408.5	2053088.2	2895922.9	3019601.6	2216507.9
Cycles Desprotegido	1630762.4	1487209.7	1573970.6	1798767.4	1491280.1	2061819.9	1638639.4	1638600.6
Instructions Overhead (%)	45.85	43.78	53.38	74.93	45.62	48.78	113.62	43.64
Instructions Protegido	1343178.4	1270784.3	1417661.7	1918620.4	1275358.8	1962938.9	2072129.1	1386568.1
Instructions Desprotegido	920955.57	883838.8	924279.05	1096790.7	875796.74	1319332.2	969995.93	965318.47

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	readlink	realpath	rm	rmdir	seq	sha1sum	sha224sum	sha256sum
Cache-Misses Overhead (%)	23.93	22.99	22.61	21.64	23.53	2.2	1.12	1.9
Cache-Misses Protegido	3076.09	3139.24	4950.25	3356.63	3388.58	19851.04	19470.05	19182.33
Cache-Misses Desprotegido	2482.14	2552.42	4037.45	2759.51	2743.08	19422.8	19254.69	18823.84
Cache-References Overhead (%)	25.77	25.83	28.16	23.16	26.33	7.6	8.05	7.47
Cache-References Protegido	24990.51	25089.18	33223.82	25671.64	25727.97	64249.18	63022.12	62746.21
Cache-References Desprotegido	19869.76	19939.5	25922.7	20843.52	20365.11	59712.6	58326.35	58384.22
Branch-Misses Overhead (%)	47.57	50.25	55.24	34.16	49.98	21.63	22.11	21.11
Branch-Misses Protegido	12444.07	12705.77	16377.07	11678.86	12885.3	17928.41	17858.69	17830.1
Branch-Misses Desprotegido	8432.85	8456.54	10549.24	8705.47	8591.21	14740.58	14625.34	14721.63
Branches Overhead (%)	62.33	65.88	77.59	42.63	57.11	15.38	15.59	15.59
Branches Protegido	266960.75	274127.03	363838.24	237335.16	290467.21	575727.61	580895.9	580996.12
Branches Desprotegido	164459.93	165254.78	204873.94	166400.49	184886.72	498977.08	502546.51	502653.41
Stalled-Cycles-Frontend Overhead (%)	43.37	44.75	49.14	32.43	41.46	9.26	12.42	12.48
Stalled-Cycles-Frontend Protegido	1553834.7	1592094.1	2198882.1	1507435.8	1626746.9	3542877.9	3527717.1	3523719.2
Stalled-Cycles-Frontend Desprotegido	1083814.3	1099865.2	1474366.6	1138323.3	1149963.8	3242507.2	3138088.2	3132707.9
Task-Clock Overhead (%)	49.09	50.91	57.75	28.81	45.76	12.58	13.84	13.12
Task-Clock Protegido	0.82	0.83	1.12	0.76	0.86	1.7	1.81	1.81
Task-Clock Desprotegido	0.55	0.55	0.71	0.59	0.59	1.51	1.59	1.6
Cycles Overhead (%)	50.46	52.41	58.43	36.61	47.77	11.67	13.93	13.99
Cycles Protegido	2273327.2	2330496.1	3189875.3	2143743.2	2404153.6	4991678.1	5070053.0	5061762.0
Cycles Desprotegido	1510881.6	1529097.5	2013404.7	1569208.7	1626965.9	4469860.1	4450182.5	4440396.5
Instructions Overhead (%)	64.68	68.09	79.79	44.69	60.01	16.13	16.38	16.42
Instructions Protegido	1484305.4	1521334.4	2034542.5	1311243.5	1594730.0	3159677.2	3335877.2	3328153.9
Instructions Desprotegido	901322.2	905048.35	1131618.6	906264.01	996663.76	2720695.5	2866439.2	2858822.2

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	sha384sum	sha512sum	shred	shuf	sleep	sort	split	stat
Cache-Misses Overhead (%)	2.88	-1.13	21.98	27.15	31.23	18.79	18.06	31.98
Cache-Misses Protegido	19126.23	18721.26	4193.91	3409.95	2604.31	16802.2	5418.8	3398.2
Cache-Misses Desprotegido	18590.9	18935.43	3438.07	2681.93	1984.58	14144.98	4589.73	2574.77
Cache-References Overhead (%)	7.99	7.24	23.07	29.1	25.64	20.59	21.9	30.4
Cache-References Protegido	62636.69	62924.55	35197.98	26744.52	25109.38	71219.36	38696.99	26662.41
Cache-References Desprotegido	58004.0	58677.8	28600.29	20716.04	19984.94	59058.59	31744.9	20445.9
Branch-Misses Overhead (%)	21.5	20.62	39.29	55.23	36.15	55.74	40.08	67.09
Branch-Misses Protegido	17871.58	17836.48	16433.08	13805.37	11658.22	21160.86	15678.04	14299.7
Branch-Misses Desprotegido	14708.53	14787.56	11797.65	8893.58	8562.72	13587.19	11192.11	8558.12
Branches Overhead (%)	15.53	15.49	56.17	70.14	49.02	63.01	51.11	89.34
Branches Protegido	579390.01	580814.01	309556.77	322911.22	239041.65	896304.19	354695.77	322604.1
Branches Desprotegido	501517.0	502896.24	198213.35	189791.08	160411.94	549861.37	234721.2	170383.85
Stalled-Cycles-Frontend Overhead (%)	14.24	11.16	39.23	49.73	37.22	38.76	33.05	58.13
Stalled-Cycles-Frontend Protegido	3544649.6	3511661.5	2025307.0	1773051.4	1449757.1	5850633.3	2270521.0	1776154.3
Stalled-Cycles-Frontend Desprotegido	3102760.1	3159117.9	1454670.6	1184192.0	1056528.8	4216304.5	1706520.0	1123230.1
Task-Clock Overhead (%)	13.04	11.87	41.77	61.67	36.07	46.53	38.75	66.07
Task-Clock Protegido	1.82	1.79	1.12	0.97	0.83	2.96	1.11	0.93
Task-Clock Desprotegido	1.61	1.6	0.79	0.6	0.61	2.02	0.8	0.56
Cycles Overhead (%)	15.35	13.02	45.12	56.72	41.93	46.92	38.92	69.17
Cycles Protegido	5069351.3	5033151.5	2885548.9	2640918.1	2090164.1	8341450.2	3240537.9	2647994.4
Cycles Desprotegido	4394917.4	4453317.0	1988429.8	1685112.8	1472624.3	5677733.1	2332661.0	1565282.8
Instructions Overhead (%)	16.27	16.11	57.85	70.68	51.03	67.17	52.13	92.34
Instructions Protegido	3301644.7	3296581.3	1754063.5	1791345.7	1324415.4	5270795.6	1998986.8	1799838.0
Instructions Desprotegido	2839575.5	2839197.4	1111202.4	1049541.5	876932.82	3152997.5	1314039.8	935763.41

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	stdbuf	sum	sync	tac	tail	tee	test	touch
Cache-Misses Overhead (%)	23.42	21.14	21.91	26.85	36.47	18.74	26.81	23.01
Cache-Misses Protegido	3061.31	3732.21	3113.02	4566.4	3935.96	3599.19	2918.43	3328.56
Cache-Misses Desprotegido	2480.48	3080.92	2553.6	3599.81	2884.07	3031.12	2301.46	2706.0
Cache-References Overhead (%)	26.76	22.33	24.56	27.27	31.88	23.92	27.18	23.25
Cache-References Protegido	25371.83	29228.69	25153.45	31972.41	28741.45	28854.19	23289.66	28872.44
Cache-References Desprotegido	20015.59	23894.12	20194.18	25122.35	21794.11	23285.34	18313.05	23425.13
Branch-Misses Overhead (%)	38.98	40.29	36.61	61.5	73.16	40.47	50.26	45.83
Branch-Misses Protegido	11881.05	12888.83	11591.48	16248.17	15485.12	13123.1	11589.44	13948.18
Branch-Misses Desprotegido	8548.68	9187.42	8485.28	10060.57	8942.92	9342.14	7712.76	9564.64
Branches Overhead (%)	49.74	50.96	42.69	88.3	94.74	49.6	55.92	60.37
Branches Protegido	248937.24	292719.76	233521.87	371424.96	334516.73	278228.5	240783.63	291355.12
Branches Desprotegido	166241.25	193903.14	163652.97	197252.22	171771.89	185980.02	154427.22	181679.7
Stalled-Cycles-Frontend Overhead (%)	35.87	34.92	31.52	51.87	62.62	33.55	41.49	56.29
Stalled-Cycles-Frontend Protegido	1515125.3	1780133.2	1472136.3	2134986.5	1937510.8	1737008.4	1430899.2	1780240.5
Stalled-Cycles-Frontend Desprotegido	1115106.2	1319417.3	1119291.8	1405778.5	1191411.5	1300655.7	1011338.1	1139094.6
Task-Clock Overhead (%)	48.21	43.08	35.09	60.27	72.88	37.31	44.9	60.0
Task-Clock Protegido	0.83	0.93	0.77	1.17	1.02	0.92	0.71	0.88
Task-Clock Desprotegido	0.56	0.65	0.57	0.73	0.59	0.67	0.49	0.55
Cycles Overhead (%)	41.14	40.39	35.89	63.38	73.43	39.14	47.44	61.2
Cycles Protegido	2185147.6	2573126.9	2100948.1	3135203.3	2843525.5	2488648.8	2078909.2	2568570.9
Cycles Desprotegido	1548164.1	1832814.2	1546030.7	1918957.2	1639608.4	1788610.3	1409985.4	1593413.9
Instructions Overhead (%)	51.42	51.76	43.69	91.5	97.6	50.98	58.29	70.6
Instructions Protegido	1384850.8	1653070.3	1292010.2	2068079.5	1858211.6	1545059.8	1335548.9	1618295.3
Instructions Desprotegido	914560.84	1089244.1	899173.54	1079925.5	940391.1	1023330.8	843758.97	948581.53

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O2
Programa	tr	true	truncate	isort	tty	uname	unexpand	uniq
Cache-Misses Overhead (%)	25.96	27.26	19.64	24.69	24.09	19.79	18.4	19.75
Cache-Misses Protegido	3424.13	2497.32	3232.48	3447.02	3017.7	2888.07	3615.89	3524.83
Cache-Misses Desprotegido	2718.5	1962.39	2701.84	2764.45	2431.87	2410.97	3053.94	2943.55
Cache-References Overhead (%)	27.27	26.15	23.36	27.06	25.01	22.92	23.16	20.27
Cache-References Protegido	26161.69	21963.25	25746.09	26579.9	24409.91	23392.6	27384.01	30927.78
Cache-References Desprotegido	20555.54	17410.62	20870.87	20919.17	19526.42	19030.15	22234.99	25716.21
Branch-Misses Overhead (%)	46.43	37.18	35.54	41.62	37.62	36.88	33.71	35.02
Branch-Misses Protegido	12463.47	10208.86	11919.49	12256.3	11315.82	10824.13	12201.44	13852.45
Branch-Misses Desprotegido	8511.31	7441.85	8794.04	8654.6	8222.74	7907.62	9125.34	10259.64
Branches Overhead (%)	60.88	42.17	44.33	50.86	43.32	43.14	42.91	38.32
Branches Protegido	266974.2	205753.35	239840.72	255833.1	230677.21	223632.96	253812.18	300627.31
Branches Desprotegido	165942.25	144726.12	166177.77	169578.16	160948.17	156234.02	177602.0	217340.16
Stalled-Cycles-Frontend Overhead (%)	54.06	31.99	31.71	33.78	34.73	31.99	33.0	28.98
Stalled-Cycles-Frontend Protegido	1725400.5	1251657.8	1513056.6	1579536.2	1427209.0	1377981.9	1611137.5	1790586.1
Stalled-Cycles-Frontend Desprotegido	1119937.0	948267.84	1148813.3	1180667.5	1059324.8	1044013.1	1211423.7	1388272.4
Task-Clock Overhead (%)	58.49	33.96	36.84	42.11	37.04	34.62	35.59	33.33
Task-Clock Protegido	0.84	0.71	0.78	0.81	0.74	0.7	0.8	0.92
Task-Clock Desprotegido	0.53	0.53	0.57	0.57	0.54	0.52	0.59	0.69
Cycles Overhead (%)	58.54	36.25	36.46	39.7	38.57	36.48	37.05	32.86
Cycles Protegido	2474383.6	1807339.4	2158323.8	2269099.5	2045022.4	1978463.3	2292370.4	2586816.8
Cycles Desprotegido	1560683.7	1326472.7	1581675.4	1624223.3	1475779.6	1449646.3	1672644.6	1947003.5
Instructions Overhead (%)	67.64	43.93	46.12	52.51	44.91	44.96	44.96	39.92
Instructions Protegido	1530268.9	1142548.8	1327390.8	1419064.0	1272808.3	1237933.9	1404015.7	1654462.6
Instructions Desprotegido	912821.67	793817.1	908421.39	930473.18	878333.45	853963.29	968577.85	1182420.8

Otimização	-O2	-O2	-O2	-O2	-O2	-O2	-O2	-O3
Programa	unlink	users	vdirt	wc	who	whoami	yes	l
Cache-Misses Overhead (%)	18.31	22.28	31.95	9.7	22.62	18.89	16.75	21.84
Cache-Misses Protegido	3175.13	3203.47	4366.36	7098.56	3310.7	3236.65	4518.86	2977.84
Cache-Misses Desprotegido	2683.78	2619.69	3309.04	6470.92	2699.94	2722.35	3870.61	2444.15
Cache-References Overhead (%)	23.25	25.42	32.29	21.34	24.82	22.62	21.51	24.91
Cache-References Protegido	24652.26	25152.75	29432.83	32770.93	25311.3	25178.28	30498.06	24113.47
Cache-References Desprotegido	20001.77	20055.08	22248.56	27007.75	20278.19	20533.35	25099.49	19305.22
Branch-Misses Overhead (%)	35.03	35.63	79.71	46.11	45.14	34.45	37.82	41.82
Branch-Misses Protegido	11382.74	11492.95	17482.05	13924.44	12671.51	11536.91	12607.19	11626.39
Branch-Misses Desprotegido	8429.5	8473.98	9727.93	9530.32	8730.45	8580.82	9147.62	8197.71
Branches Overhead (%)	43.36	47.83	121.31	45.95	55.02	42.33	40.99	50.29
Branches Protegido	233812.14	243281.0	428578.35	366428.5	266669.98	235307.27	296399.44	246243.08
Branches Desprotegido	163097.05	164562.6	193658.64	251067.53	172021.1	165329.11	210227.9	163850.58
Stalled-Cycles-Frontend Overhead (%)	30.28	38.33	76.58	32.09	39.86	31.92	30.91	36.36
Stalled-Cycles-Frontend Protegido	1470471.5	1508108.2	2191278.2	2144984.3	1570662.9	1481853.1	1819500.6	1449999.1
Stalled-Cycles-Frontend Desprotegido	1128741.4	1090213.2	1240975.2	1623925.0	1123033.3	1123336.5	1389900.0	1063357.9
Task-Clock Overhead (%)	30.36	44.44	95.0	36.14	42.86	34.55	28.77	41.07
Task-Clock Protegido	0.73	0.78	1.17	1.13	0.8	0.74	0.94	0.79
Task-Clock Desprotegido	0.56	0.54	0.6	0.83	0.56	0.55	0.73	0.56
Cycles Overhead (%)	35.17	42.55	92.12	37.06	46.01	35.96	34.85	41.73
Cycles Protegido	2101342.3	2162380.8	3332463.7	3117093.0	2285087.1	2114920.2	2616552.9	2108257.5
Cycles Desprotegido	1554635.1	1516882.5	1734543.8	2274242.0	1565069.3	1555500.7	1940399.6	1487539.8
Instructions Overhead (%)	45.11	50.3	126.84	47.0	57.57	43.3	42.37	52.01
Instructions Protegido	1298288.5	1352190.3	2385022.7	2028124.4	1476788.0	1302851.8	1646389.4	1360284.7
Instructions Desprotegido	894668.22	899648.55	1051417.1	1379641.8	937201.87	909168.06	1156378.6	894846.3

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	b2sum	base32	base64	basename	basenc	cat	chgrp	chmod
Cache-Misses Overhead (%)	17.43	25.37	19.67	22.35	34.32	17.11	21.79	26.48
Cache-Misses Protegido	3588.18	3820.48	3279.49	3129.39	4240.51	4307.65	5054.8	4166.93
Cache-Misses Desprotegido	3055.57	3047.3	2740.52	2557.66	3157.03	3678.35	4150.59	3294.47
Cache-References Overhead (%)	22.89	22.59	24.57	22.87	31.26	20.87	23.58	29.26
Cache-References Protegido	27010.24	26935.45	25000.3	23889.06	29399.91	27510.19	35239.05	30397.85
Cache-References Desprotegido	21979.69	21972.47	20068.72	19443.31	22398.99	22760.85	28516.04	23516.32
Branch-Misses Overhead (%)	40.93	41.39	40.92	38.14	78.98	41.19	45.01	59.77
Branch-Misses Protegido	12815.86	12619.92	11965.61	11172.04	16141.04	12156.55	17402.78	15181.37
Branch-Misses Desprotegido	9093.83	8925.45	8491.05	8087.53	9018.33	8610.35	12001.2	9501.74
Branches Overhead (%)	50.61	48.46	47.46	43.49	116.62	45.6	62.55	81.74
Branches Protegido	273714.31	250796.09	244307.16	227240.87	390883.81	264663.12	390903.07	337949.7
Branches Desprotegido	181739.62	168936.73	165673.15	158367.55	180444.08	181779.93	240479.65	185951.6
Stalled-Cycles-Frontend Overhead (%)	37.56	31.93	34.68	32.24	67.8	30.76	41.86	52.49
Stalled-Cycles-Frontend Protegido	1675729.0	1590076.3	1522806.0	1423455.6	2098440.0	1693823.6	2257292.1	1990247.8
Stalled-Cycles-Frontend Desprotegido	1218144.8	1205244.8	1130679.7	1076440.6	1250571.9	1295362.9	1591228.9	1305204.5
Task-Clock Overhead (%)	44.07	37.5	33.33	35.85	83.05	34.92	50.0	60.0
Task-Clock Protegido	0.85	0.77	0.76	0.72	1.08	0.85	1.17	1.04
Task-Clock Desprotegido	0.59	0.56	0.57	0.53	0.59	0.63	0.78	0.65
Cycles Overhead (%)	43.02	37.56	39.6	36.78	83.28	35.82	49.29	62.24
Cycles Protegido	2415231.5	2266503.4	2182570.9	2034437.4	3147621.8	2411720.9	3300860.8	2912418.8
Cycles Desprotegido	1688776.9	1647659.7	1563420.8	1487417.9	1717411.9	1775660.5	2211083.3	1795081.9
Instructions Overhead (%)	53.73	49.94	49.19	45.31	120.26	47.07	64.5	84.36
Instructions Protegido	1521199.3	1386697.4	1356822.0	1257844.6	2171443.9	1467238.2	2165215.5	1885915.6
Instructions Desprotegido	989519.8	924842.62	909462.64	865631.13	985835.94	997629.95	1316274.9	1022949.1

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	chown	chroot	cksum	comm	cp	csplit	cut	date
Cache-Misses Overhead (%)	23.68	22.65	7.69	21.05	24.71	30.33	24.11	22.38
Cache-Misses Protegido	4310.54	3110.84	21429.12	3502.78	5499.32	4508.67	3745.19	4028.4
Cache-Misses Desprotegido	3485.21	2536.4	19898.52	2893.76	4409.71	3459.3	3017.68	3291.69
Cache-References Overhead (%)	25.84	24.53	11.93	24.48	28.54	27.3	23.37	27.63
Cache-References Protegido	31458.74	24908.71	71492.58	26785.09	34323.83	33772.62	31018.2	31364.14
Cache-References Desprotegido	24998.56	20002.49	63872.12	21517.46	26702.58	26529.92	25142.07	24574.08
Branch-Misses Overhead (%)	51.01	35.94	35.63	39.2	66.65	62.53	46.6	51.77
Branch-Misses Protegido	16104.67	11451.85	23474.67	12406.18	17956.3	16790.25	14640.19	15542.93
Branch-Misses Desprotegido	10664.93	8424.29	17307.55	8912.42	10774.81	10330.31	9986.38	10241.13
Branches Overhead (%)	68.83	47.75	31.23	48.14	97.03	92.64	56.98	67.16
Branches Protegido	359938.94	244956.97	792151.35	256032.5	425052.78	396931.67	331467.94	332308.24
Branches Desprotegido	213201.8	165793.51	603642.39	172833.09	215728.99	206045.2	211153.26	198791.57
Stalled-Cycles-Frontend Overhead (%)	44.94	35.14	22.26	34.93	57.34	54.53	40.48	45.85
Stalled-Cycles-Frontend Protegido	2033965.9	1501844.3	4328967.0	1606218.7	2402701.2	2212456.4	1901205.2	1955733.4
Stalled-Cycles-Frontend Desprotegido	1403333.5	1111355.1	3540651.1	1190419.0	1527057.2	1431732.2	1353396.0	1340957.9
Task-Clock Overhead (%)	51.43	34.48	26.82	42.86	67.57	63.38	46.27	52.31
Task-Clock Protegido	1.06	0.78	2.27	0.8	1.24	1.16	0.98	0.99
Task-Clock Desprotegido	0.7	0.58	1.79	0.56	0.74	0.71	0.67	0.65
Cycles Overhead (%)	53.52	40.02	25.94	40.1	69.61	66.62	46.89	53.82
Cycles Protegido	2999428.6	2158876.2	6348868.7	2297243.3	3550005.6	3284258.5	2786027.9	2854310.1
Cycles Desprotegido	1953804.9	1541872.5	5041141.2	1639684.0	2093039.1	1971152.4	1896706.5	1855651.1
Instructions Overhead (%)	71.54	49.72	32.95	50.72	99.71	95.49	59.3	69.97
Instructions Protegido	1999799.8	1360500.0	4330406.5	1420140.5	2376856.1	2216960.6	1833358.6	1839986.6
Instructions Desprotegido	1165804.9	908667.74	3257267.2	942247.38	1190174.1	1134052.1	1150892.4	1082519.9

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	dd	df	dir	dircolors	dirname	du	echo	expand	
Cache-Misses Overhead (%)	22.85	29.87	28.89	22.05	22.55	39.1	20.02	26.55	
Cache-Misses Protegido	4350.34	4140.93	4247.06	3300.22	3167.82	5055.09	2969.56	3533.49	
Cache-Misses Desprotegido	3541.15	3188.46	3295.1	2704.09	2584.96	3634.04	2474.25	2792.26	
Cache-References Overhead (%)	26.82	31.11	30.81	25.49	22.71	35.34	24.25	23.81	
Cache-References Protegido	30274.52	29481.81	29074.96	25842.6	24598.88	33052.33	23784.58	27385.48	
Cache-References Desprotegido	23871.2	22486.48	22227.38	20592.63	20046.54	24420.83	19141.98	22118.61	
Branch-Misses Overhead (%)	48.33	69.35	78.79	38.85	35.55	92.37	38.49	38.62	
Branch-Misses Protegido	14242.76	16847.64	17268.46	11985.26	11292.84	19260.21	11173.32	12536.46	
Branch-Misses Desprotegido	9602.36	9948.3	9658.78	8631.55	8330.88	10012.3	8068.18	9043.47	
Branches Overhead (%)	63.95	87.91	117.27	51.66	43.41	142.57	46.39	44.68	
Branches Protegido	310108.08	393873.62	420465.98	256121.76	231988.62	491380.2	229547.97	258807.01	
Branches Desprotegido	189150.86	209609.18	193518.94	168882.0	161763.03	202574.33	156810.42	178883.27	
Stalled-Cycles-Frontend Overhead (%)	42.38	60.99	69.78	36.79	33.39	84.73	31.97	31.9	
Stalled-Cycles-Frontend Protegido	1909548.0	2048318.2	2146175.6	1557568.0	1467484.1	2554312.4	1408732.6	1611236.9	
Stalled-Cycles-Frontend Desprotegido	1341202.2	1272346.0	1264078.2	1138643.9	1100148.1	1382699.5	1067494.7	1221547.0	
Task-Clock Overhead (%)	46.15	74.6	85.71	40.35	37.74	101.47	46.15	33.33	
Task-Clock Protegido	0.95	1.1	1.17	0.8	0.73	1.37	0.76	0.8	
Task-Clock Desprotegido	0.65	0.63	0.63	0.57	0.53	0.68	0.52	0.6	
Cycles Overhead (%)	49.86	72.51	85.97	42.33	37.59	103.03	37.31	36.78	
Cycles Protegido	2752865.0	3073691.8	3267343.3	2245144.9	2092133.9	3882420.9	2026984.5	2306042.4	
Cycles Desprotegido	1836967.8	1781752.2	1756876.6	1577459.7	1520574.1	1912202.9	1476162.4	1685916.6	
Instructions Overhead (%)	66.32	94.74	122.39	53.7	45.62	145.84	48.24	46.85	
Instructions Protegido	1728656.2	2136145.0	2339059.9	1418851.2	1284763.6	2746016.2	1271226.7	1432624.7	
Instructions Desprotegido	1039385.0	1096942.3	1051760.8	923115.64	882290.45	1116974.4	857527.01	975588.29	

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	expr	factor	false	fmt	fold	ginstall	groups	head
Cache-Misses Overhead (%)	60.47	30.71	26.1	22.28	16.47	32.59	14.11	18.19
Cache-Misses Protegido	4240.08	3979.69	2891.89	3512.36	3959.41	4889.49	3263.08	3009.1
Cache-Misses Desprotegido	2642.21	3044.73	2293.32	2872.32	3399.42	3687.62	2859.7	2546.06
Cache-References Overhead (%)	44.48	34.26	25.96	24.89	20.79	33.16	23.46	23.3
Cache-References Protegido	28943.25	28153.61	23702.22	27171.65	32510.84	33425.55	26406.31	26675.02
Cache-References Desprotegido	20033.15	20970.24	18817.84	21757.28	26914.28	25102.62	21388.92	21634.62
Branch-Misses Overhead (%)	102.87	87.84	42.78	40.18	34.34	84.97	32.95	42.93
Branch-Misses Protegido	17674.16	16752.08	11071.06	12788.34	14228.63	19187.35	12064.22	12605.56
Branch-Misses Desprotegido	8711.89	8918.09	7754.13	9123.12	10591.36	10373.0	9074.15	8819.32
Branches Overhead (%)	171.14	119.07	42.2	47.93	34.93	121.27	40.39	51.01
Branches Protegido	463511.71	403640.72	222520.99	271037.86	363078.93	463215.7	250096.15	263071.35
Branches Desprotegido	170951.4	184252.01	156483.89	183215.15	269082.67	209343.72	178139.62	174207.93
Stalled-Cycles-Frontend Overhead (%)	103.92	73.06	40.19	35.14	29.33	73.66	29.53	34.88
Stalled-Cycles-Frontend Protegido	2287087.5	2466304.8	1430067.4	1626760.1	1950846.3	2432652.8	1542785.3	1581943.4
Stalled-Cycles-Frontend Desprotegido	1121539.4	1425148.9	1020101.2	1203789.7	1508420.7	1400786.1	1191101.5	1172859.5
Task-Clock Overhead (%)	122.41	86.11	37.74	40.98	32.91	87.14	40.68	40.0
Task-Clock Protegido	1.29	1.34	0.73	0.86	1.05	1.31	0.83	0.84
Task-Clock Desprotegido	0.58	0.72	0.53	0.61	0.79	0.7	0.59	0.6
Cycles Overhead (%)	126.41	85.17	42.38	40.13	31.35	89.19	33.92	40.81
Cycles Protegido	3536004.8	3831060.5	2035994.1	2350074.2	2915123.0	3680765.0	2213424.4	2288900.9
Cycles Desprotegido	1561737.9	2068891.9	1429988.6	1677068.3	2219337.0	1945524.1	1652739.0	1625484.2
Instructions Overhead (%)	177.17	112.44	44.65	49.46	36.45	124.42	42.62	53.11
Instructions Protegido	2585995.8	2845469.4	1243852.1	1498475.9	1967260.6	2584638.7	1384717.6	1447216.5
Instructions Desprotegido	933011.48	1339392.5	859883.51	1002607.4	1441720.9	1151677.6	970912.7	945238.83

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	hostid	id	join	link	ln	logname	ls	md5sum
Cache-Misses Overhead (%)	15.53	16.14	20.2	22.47	26.27	18.72	37.83	3.41
Cache-Misses Protegido	3259.18	3952.83	4033.98	3210.67	3665.04	3178.57	4695.98	20434.66
Cache-Misses Desprotegido	2821.17	3403.38	3356.07	2621.54	2902.49	2677.45	3407.04	19761.5
Cache-References Overhead (%)	22.32	21.72	25.34	22.27	26.42	24.01	38.5	9.09
Cache-References Protegido	24986.99	29948.2	32904.02	24657.94	27374.65	24945.94	29925.1	69707.15
Cache-References Desprotegido	20427.09	24603.35	26252.79	20166.89	21654.38	20115.86	21606.96	63898.1
Branch-Misses Overhead (%)	34.34	36.05	45.75	36.36	62.02	39.03	88.22	22.39
Branch-Misses Protegido	11545.58	14073.99	15639.26	11565.53	14747.3	11764.79	16742.2	18732.75
Branch-Misses Desprotegido	8594.54	10344.34	10730.25	8481.81	9102.31	8461.87	8895.01	15305.22
Branches Overhead (%)	41.93	41.15	55.07	44.41	89.0	45.96	133.46	15.68
Branches Protegido	235381.54	296806.48	351885.92	236004.75	319105.07	238978.28	425225.62	610419.81
Branches Desprotegido	165839.84	210273.25	226926.93	163422.21	168843.04	163731.46	182138.31	527669.51
Stalled-Cycles-Frontend Overhead (%)	32.03	30.7	40.75	35.47	54.82	32.99	82.27	11.76
Stalled-Cycles-Frontend Protegido	1482251.1	1837667.7	2034413.9	1511333.8	1825171.2	1499884.7	2228820.9	3771374.7
Stalled-Cycles-Frontend Desprotegido	1122670.0	1406017.8	1445448.9	1115607.2	1178891.5	1127795.3	1222830.8	3374636.5
Task-Clock Overhead (%)	41.82	33.82	49.28	32.73	61.4	39.62	96.61	13.5
Task-Clock Protegido	0.78	0.91	1.03	0.73	0.92	0.74	1.16	1.85
Task-Clock Desprotegido	0.55	0.68	0.69	0.55	0.57	0.53	0.59	1.63
Cycles Overhead (%)	36.26	34.88	46.61	39.35	66.0	37.91	99.17	13.56
Cycles Protegido	2115322.4	2631522.4	2970589.7	2148131.9	2688522.0	2143819.6	3376743.8	5298425.5
Cycles Desprotegido	1552374.0	1950992.2	2026234.0	1541564.7	1619604.4	1554533.0	1695370.8	4665950.0
Instructions Overhead (%)	44.3	42.97	57.63	46.35	91.93	47.8	138.01	16.41
Instructions Protegido	1302765.2	1640049.5	1943490.4	1309570.0	1776511.0	1325823.5	2374736.8	3294144.7
Instructions Desprotegido	902798.52	1147133.4	1232913.0	894834.71	925583.28	897038.87	997748.43	2829859.6

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	mkdir	mkfifo	mknod	mktemp	mv	nl	nproc
Cache-Misses Overhead (%)	16.99	20.18	19.25	15.97	39.52	31.57	17.7
Cache-Misses Protegido	4190.06	3261.82	3174.07	4428.85	3706.9	3649.95	3002.54
Cache-Misses Desprotegido	3581.41	2714.06	2661.64	3819.01	2656.87	2774.23	2550.92
Cache-References Overhead (%)	23.32	23.44	23.9	22.21	34.14	29.21	23.5
Cache-References Protegido	29144.31	25475.7	25093.31	29119.89	30210.97	28083.24	25316.2
Cache-References Desprotegido	23632.59	20638.01	20252.32	23828.51	22522.4	21734.38	20498.78
Branch-Misses Overhead (%)	41.2	38.78	39.1	35.65	85.19	67.01	37.8
Branch-Misses Protegido	13784.15	12006.02	11849.36	13460.14	17767.67	15103.13	11849.76
Branch-Misses Desprotegido	9762.37	8650.96	8518.74	9922.82	9594.5	9043.24	8599.21
Branches Overhead (%)	54.67	48.31	47.93	48.69	128.85	94.36	46.38
Branches Protegido	276481.31	243023.2	243679.97	263618.85	419086.34	350762.54	244294.79
Branches Desprotegido	178757.74	163863.54	164721.77	177293.96	183130.91	180472.79	166894.49
Stalled-Cycles-Frontend Overhead (%)	35.34	34.45	34.48	31.69	78.93	58.45	31.17
Stalled-Cycles-Frontend Protegido	1796346.0	1526608.8	1515648.0	1754396.7	2188152.0	1904759.6	2627940.9
Stalled-Cycles-Frontend Desprotegido	1327279.0	1135418.5	1127018.7	1332238.1	1222939.9	1202082.8	2022189.2
Task-Clock Overhead (%)	39.06	39.29	42.59	35.82	91.18	62.3	34.48
Task-Clock Protegido	0.89	0.78	0.77	0.91	1.3	0.99	0.78
Task-Clock Desprotegido	0.64	0.56	0.54	0.67	0.68	0.61	0.58
Cycles Overhead (%)	41.69	39.68	39.61	37.29	95.56	70.59	36.73
Cycles Protegido	2543609.8	2182510.5	2172316.4	2469607.9	3322065.8	2846528.5	2145055.4
Cycles Desprotegido	1795209.7	1562466.8	1555952.3	1798853.8	1698726.3	1668672.6	1568792.4
Instructions Overhead (%)	56.48	50.51	49.65	50.4	132.41	97.5	48.14
Instructions Protegido	1538336.3	1349300.6	1352172.6	1465523.1	2346297.2	1950198.1	1354982.6
Instructions Desprotegido	983104.76	896466.66	903543.94	974390.06	1009539.6	987429.58	914651.04

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	numfmt	od	paste	pathchk	pinky	pr	printenv	printf
Cache-Misses Overhead (%)	24.78	23.27	23.96	21.4	21.68	24.75	22.78	26.13
Cache-Misses Protegido	3410.97	3726.84	3230.35	3043.45	3237.63	4006.4	3070.64	3674.32
Cache-Misses Desprotegido	2733.51	3023.29	2605.86	2506.96	2660.74	3211.64	2500.95	2913.22
Cache-References Overhead (%)	26.67	28.08	24.49	25.16	27.56	28.79	23.84	22.8
Cache-References Protegido	27024.91	28484.9	28009.51	24203.39	25651.86	30361.62	24639.34	32678.56
Cache-References Desprotegido	21335.59	22239.22	22498.71	19337.33	20109.37	23574.48	19896.07	26610.38
Branch-Misses Overhead (%)	54.18	48.65	36.07	37.37	43.83	47.71	39.26	46.4
Branch-Misses Protegido	13815.61	13812.3	12343.89	11258.97	12443.41	14816.43	11433.65	15491.63
Branch-Misses Desprotegido	8960.6	9292.11	9071.92	8196.05	8651.61	10030.7	8210.13	10581.37
Branches Overhead (%)	69.91	62.58	44.74	43.82	52.08	64.11	39.8	48.77
Branches Protegido	303419.22	304828.44	244260.72	233223.98	256532.57	345322.73	230850.49	362727.78
Branches Desprotegido	178573.94	187488.68	168752.84	162162.32	168681.12	210424.43	165129.41	243823.52
Stalled-Cycles-Frontend Overhead (%)	47.5	43.92	33.01	31.99	35.86	44.96	31.97	38.31
Stalled-Cycles-Frontend Protegido	1735720.3	1787517.1	1585145.5	1419150.3	1555549.4	1892403.3	1437587.7	1995044.9
Stalled-Cycles-Frontend Desprotegido	1176739.5	1242010.7	1191708.5	1075165.4	1144998.8	1305468.5	1089293.5	1442496.0
Task-Clock Overhead (%)	60.34	50.0	36.07	34.55	42.86	51.56	32.14	41.1
Task-Clock Protegido	0.93	0.96	0.83	0.74	0.8	0.97	0.74	1.03
Task-Clock Desprotegido	0.58	0.64	0.61	0.55	0.56	0.64	0.56	0.73
Cycles Overhead (%)	55.79	50.82	37.39	36.78	41.87	52.72	36.33	43.24
Cycles Protegido	2550421.5	2607456.7	2246933.2	2045772.7	2246879.4	2780845.0	2056004.6	2944803.5
Cycles Desprotegido	1637070.6	1728850.6	1635419.5	1495671.5	1583706.6	1820855.8	1508110.5	2055871.4
Instructions Overhead (%)	72.72	64.41	46.86	45.37	54.14	66.84	44.71	51.02
Instructions Protegido	1683598.3	1687875.0	1357923.6	1291731.8	1426907.7	1844488.8	1272935.8	1992723.6
Instructions Desprotegido	974781.2	1026623.6	924619.07	888608.99	925747.49	1105547.5	879639.84	1319522.6

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	ptx	pwd	readlink	realpath	rm	rmdir	seq	sha1sum
Cache-Misses Overhead (%)	35.45	21.88	25.85	24.09	21.67	19.74	21.88	1.5
Cache-Misses Protegido	3997.85	3150.43	3094.38	3193.76	4924.49	3342.29	3354.97	19435.79
Cache-Misses Desprotegido	2951.55	2584.84	2458.71	2573.69	4047.55	2791.19	2752.67	19149.46
Cache-References Overhead (%)	32.98	23.34	26.89	26.51	27.64	23.78	25.82	7.87
Cache-References Protegido	28286.71	26490.6	25239.69	25176.23	33084.94	25644.08	25647.33	64084.89
Cache-References Desprotegido	21271.84	21477.89	19891.13	19901.26	25920.42	20717.12	20384.04	59408.46
Branch-Misses Overhead (%)	73.75	36.16	55.3	50.94	52.88	37.03	48.64	21.54
Branch-Misses Protegido	16032.37	11935.77	13092.04	12759.67	16129.95	11919.92	12767.04	17926.72
Branch-Misses Desprotegido	9227.35	8765.94	8430.24	8453.29	10550.56	8698.52	8589.48	14750.12
Branches Overhead (%)	107.16	42.55	66.55	64.65	75.95	42.66	55.67	15.27
Branches Protegido	367909.97	251513.74	274295.42	271586.34	360956.44	237151.38	287948.33	575649.03
Branches Desprotegido	177598.99	176444.15	164694.67	164945.7	205143.8	166231.53	184974.99	499411.85
Stalled-Cycles-Frontend Overhead (%)	69.95	29.79	45.86	44.15	47.36	32.95	39.52	11.99
Stalled-Cycles-Frontend Protegido	1996987.3	1539717.3	1587936.4	1588527.4	2176596.8	1517813.2	1617078.9	3546643.9
Stalled-Cycles-Frontend Desprotegido	1175051.7	1186347.9	1088672.7	1101988.6	1477074.6	1141649.0	1159067.1	3167022.4
Task-Clock Overhead (%)	81.03	35.0	52.73	51.85	54.17	32.76	43.33	14.0
Task-Clock Protegido	1.05	0.81	0.84	0.82	1.11	0.77	0.86	1.71
Task-Clock Desprotegido	0.58	0.6	0.55	0.54	0.72	0.58	0.6	1.5
Cycles Overhead (%)	83.13	34.36	53.67	51.65	56.59	37.0	45.67	13.55
Cycles Protegido	2988670.0	2213749.1	2329392.4	2319645.3	3157760.7	2155939.5	2385190.7	4994042.5
Cycles Desprotegido	1632014.0	1647683.6	1515854.9	1529575.7	2016585.2	1573700.8	1637371.6	4398243.6
Instructions Overhead (%)	112.0	43.09	69.06	66.77	78.01	44.45	57.64	15.86
Instructions Protegido	2048266.1	1391028.2	1525648.3	1506025.5	2017699.5	1313235.5	1576102.9	3159130.8
Instructions Desprotegido	966147.07	972137.04	902413.2	903058.04	1133480.7	909108.71	999808.52	2726796.3

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	sha224sum	sha256sum	sha384sum	sha512sum	shred	shuf	sleep	sort
Cache-Misses Overhead (%)	-2.84	0.37	2.98	2.67	22.02	25.13	30.29	18.58
Cache-Misses Protegido	18768.76	18956.55	19712.96	19453.25	4135.55	3443.66	2616.67	16812.52
Cache-Misses Desprotegido	19318.24	18886.63	19143.38	18948.14	3389.28	2752.04	2008.34	14177.65
Cache-References Overhead (%)	6.59	8.44	7.67	8.42	22.96	29.48	25.38	21.66
Cache-References Protegido	62559.02	63119.23	63096.52	63128.72	35157.26	26919.8	25070.15	71675.81
Cache-References Desprotegido	58690.18	58207.14	58601.65	58226.23	28593.44	20790.81	19996.12	58917.01
Branch-Misses Overhead (%)	22.08	22.08	21.74	20.96	43.54	54.87	37.33	60.48
Branch-Misses Protegido	17962.52	17970.02	17913.09	17869.83	16916.99	13802.65	11773.03	21754.7
Branch-Misses Desprotegido	14713.2	14720.45	14714.65	14773.09	11785.34	8912.51	8573.11	13555.79
Branches Overhead (%)	15.37	15.36	15.32	15.0	59.47	72.01	48.81	68.74
Branches Protegido	579612.63	579696.19	579983.59	579754.75	315794.03	328638.0	239705.7	917922.54
Branches Desprotegido	502388.42	502521.01	502924.61	504122.17	198033.3	191057.7	161077.61	543982.71
Stalled-Cycles-Frontend Overhead (%)	12.02	13.74	11.64	14.52	37.13	51.15	37.16	41.66
Stalled-Cycles-Frontend Protegido	3513567.0	3553726.0	3516395.5	3626682.3	2026412.8	1783556.2	1456399.4	5978860.3
Stalled-Cycles-Frontend Desprotegido	3136443.6	3124442.3	3149898.3	3166729.2	1477727.6	1180000.5	1061796.7	4220580.2
Task-Clock Overhead (%)	16.13	14.65	9.55	17.11	37.35	57.38	39.34	50.0
Task-Clock Protegido	1.8	1.8	1.72	1.78	1.14	0.96	0.85	3.03
Task-Clock Desprotegido	1.55	1.57	1.57	1.52	0.83	0.61	0.61	2.02
Cycles Overhead (%)	13.63	14.87	13.38	15.42	44.12	58.54	41.94	50.54
Cycles Protegido	5057371.1	5094661.7	5038389.5	5152053.1	2902240.8	2665313.4	2100340.2	8553088.8
Cycles Desprotegido	4450899.9	4435266.2	4443803.7	4463784.5	2013775.3	1681180.0	1479688.5	5681451.7
Instructions Overhead (%)	16.04	16.12	16.21	16.25	60.18	73.45	50.85	72.4
Instructions Protegido	3331303.1	3324560.5	3297201.6	3305200.1	1785158.3	1823051.0	1329247.7	5439313.7
Instructions Desprotegido	2870879.7	2863027.2	2837338.1	2843099.9	1114448.2	1051077.6	881177.83	3155062.1

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	split	stat	stdbuf	sum	sync	tac	tail	tee
Cache-Misses Overhead (%)	17.84	34.85	21.19	17.69	15.44	30.21	34.07	19.89
Cache-Misses Protegido	5427.05	3453.88	3062.3	3617.71	3094.57	4602.06	3911.37	3629.05
Cache-Misses Desprotegido	4605.55	2561.31	2526.82	3073.99	2680.69	3534.36	2917.42	3027.06
Cache-References Overhead (%)	21.79	31.2	24.1	22.23	23.13	26.44	30.17	21.97
Cache-References Protegido	38771.55	26825.62	25144.78	29152.25	25081.78	31758.68	28531.49	28571.2
Cache-References Desprotegido	31834.69	20446.39	20260.92	23850.24	20370.08	25117.67	21918.16	23424.22
Branch-Misses Overhead (%)	41.92	75.38	37.77	39.84	35.17	61.84	58.04	35.98
Branch-Misses Protegido	15931.8	15050.8	11927.19	12877.37	11517.83	16242.07	14210.48	12757.9
Branch-Misses Desprotegido	11225.67	8581.81	8657.03	9208.36	8520.86	10035.98	8991.75	9382.33
Branches Overhead (%)	52.83	96.35	48.69	48.0	41.96	88.6	84.13	46.99
Branches Protegido	359090.95	336244.13	250646.65	289762.16	233744.08	370104.02	318125.71	275159.79
Branches Desprotegido	234956.14	171249.96	168567.28	195783.57	164659.95	196237.93	172776.57	187197.22
Stalled-Cycles-Frontend Overhead (%)	35.93	61.45	37.44	33.01	30.54	51.52	56.82	32.76
Stalled-Cycles-Frontend Protegido	2319995.1	1830873.2	1514589.7	1777615.8	1461749.3	2122681.7	1874597.8	1726503.3
Stalled-Cycles-Frontend Desprotegido	1706742.0	1133991.4	1101996.4	1336456.5	1119762.8	1400965.9	1195401.3	1300439.4
Task-Clock Overhead (%)	39.51	71.43	39.66	40.0	31.03	65.28	67.8	42.42
Task-Clock Protegido	1.13	0.96	0.81	0.91	0.76	1.19	0.99	0.94
Task-Clock Desprotegido	0.81	0.56	0.58	0.65	0.58	0.72	0.59	0.66
Cycles Overhead (%)	41.67	73.53	42.39	38.65	35.06	63.17	66.11	37.82
Cycles Protegido	3305252.6	2738833.2	2186477.8	2564520.7	2090678.5	3118992.6	2732175.2	2465331.9
Cycles Desprotegido	2333085.1	1578260.2	1535517.7	1849699.1	1548014.4	1911534.5	1644752.1	1788765.4
Instructions Overhead (%)	54.22	99.28	51.49	50.83	43.66	91.51	86.95	48.79
Instructions Protegido	2029257.3	1875981.2	1387314.2	1641494.7	1295491.2	2058715.3	1764355.7	1522289.2
Instructions Desprotegido	1315779.2	941366.88	915802.17	1088292.8	901800.69	1075009.7	943781.46	1023135.3

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	test	touch	tr	true	truncate	tsort	tty	uname
Cache-Misses Overhead (%)	23.87	21.89	25.83	25.15	19.21	16.81	20.22	20.79
Cache-Misses Protegido	2896.67	3287.03	3449.68	2502.44	3232.3	3331.35	2946.48	2913.81
Cache-Misses Desprotegido	2338.44	2696.66	2741.47	1999.57	2711.32	2852.01	2450.93	2412.23
Cache-References Overhead (%)	27.11	22.7	28.22	25.88	22.53	25.46	24.23	23.58
Cache-References Protegido	23281.14	28790.95	26392.44	22048.12	25623.25	26370.47	24147.15	23445.85
Cache-References Desprotegido	18316.33	23465.09	20584.22	17514.97	20911.21	21019.82	19437.54	18971.89
Branch-Misses Overhead (%)	47.26	41.25	49.37	39.41	33.55	38.19	39.03	40.08
Branch-Misses Protegido	11329.24	13524.76	12699.16	10400.03	11807.22	12100.76	11369.35	11091.54
Branch-Misses Desprotegido	7693.54	9574.99	8501.8	7459.83	8840.73	8756.7	8177.84	7917.84
Branches Overhead (%)	55.11	56.37	64.78	44.24	42.83	49.36	43.4	45.34
Branches Protegido	240082.22	284569.18	273443.33	209395.15	239113.72	253520.99	231933.79	228128.41
Branches Desprotegido	154785.26	181985.37	165941.78	145173.82	167413.6	169732.53	161737.05	156964.43
Stalled-Cycles-Frontend Overhead (%)	39.07	53.09	57.9	34.99	31.66	35.48	33.46	33.98
Stalled-Cycles-Frontend Protegido	1417633.5	1744674.8	1775481.5	1279171.7	1511773.4	1585667.9	1425715.8	1403020.2
Stalled-Cycles-Frontend Desprotegido	1019383.6	1139617.3	1124425.1	947636.63	1148282.1	1170413.5	1068233.0	1047203.0
Task-Clock Overhead (%)	46.94	56.36	61.11	33.96	34.48	40.35	38.89	36.54
Task-Clock Protegido	0.72	0.86	0.87	0.71	0.78	0.8	0.75	0.71
Task-Clock Desprotegido	0.49	0.55	0.54	0.53	0.58	0.57	0.54	0.52
Cycles Overhead (%)	45.37	57.71	62.96	39.03	36.14	40.68	37.7	38.6
Cycles Protegido	2062010.9	2513431.0	2549345.1	1843348.9	2154940.0	2268789.2	2049080.3	2016250.2
Cycles Desprotegido	1418501.1	1593719.1	1564393.6	1325891.7	1582921.4	1612682.6	1488036.1	1454752.6
Instructions Overhead (%)	57.39	66.67	73.0	46.11	45.23	51.28	45.33	47.14
Instructions Protegido	1331423.6	1581046.9	1578211.9	1161327.1	1325770.5	1404972.4	1285217.6	1263024.6
Instructions Desprotegido	845934.43	948593.16	912262.73	794830.6	912853.67	928714.73	884345.28	858363.92

Otimização	-O3	-O3	-O3	-O3	-O3	-O3	-O3	-O3
Programa	unexpand	uniq	unlink	users	vdirt	wc	who	whoami
Cache-Misses Overhead (%)	22.73	18.72	18.83	21.45	30.28	10.95	22.23	21.9
Cache-Misses Protegido	3671.92	3550.98	3148.99	3315.55	4298.05	7182.14	3302.39	3290.29
Cache-Misses Desprotegido	2991.95	2991.06	2649.97	2729.92	3299.08	6473.56	2701.8	2699.16
Cache-References Overhead (%)	23.67	20.7	23.12	25.94	32.31	23.11	26.12	24.64
Cache-References Protegido	27512.08	31162.1	24698.44	25421.73	29285.51	33262.09	25473.18	25226.79
Cache-References Desprotegido	22246.85	25816.84	20060.75	20185.44	22134.06	27018.11	20196.89	20238.97
Branch-Misses Overhead (%)	38.22	35.71	31.84	40.02	79.4	51.2	47.89	36.84
Branch-Misses Protegido	12613.36	13990.95	11125.73	11838.32	17277.04	14407.1	12880.1	11701.48
Branch-Misses Desprotegido	9125.59	10309.58	8439.01	8454.51	9630.23	9528.56	8709.23	8551.31
Branches Overhead (%)	44.45	40.43	43.01	47.04	117.37	51.88	59.11	43.85
Branches Protegido	257239.64	305858.27	233489.3	245445.11	421214.45	381830.37	273406.73	23882.17
Branches Desprotegido	178081.8	217802.02	163273.01	166918.99	193779.72	251400.04	171836.74	166025.93
Stalled-Cycles-Frontend Overhead (%)	33.37	30.8	33.25	31.47	72.55	35.28	40.43	35.47
Stalled-Cycles-Frontend Protegido	1618416.8	1821184.1	1476730.3	1497423.9	2143070.4	2210804.9	1589611.6	1523631.0
Stalled-Cycles-Frontend Desprotegido	1213454.2	1392366.5	1108249.6	1138962.7	1242000.5	1634234.4	1131984.8	1124710.6
Task-Clock Overhead (%)	35.0	33.33	40.0	40.74	85.48	40.96	46.43	40.0
Task-Clock Protegido	0.81	0.92	0.77	0.76	1.15	1.17	0.82	0.77
Task-Clock Desprotegido	0.6	0.69	0.55	0.54	0.62	0.83	0.56	0.55
Cycles Overhead (%)	37.59	34.85	37.17	37.45	88.0	41.04	47.43	39.23
Cycles Protegido	2306746.7	2632663.0	2104840.0	2158518.0	3264872.5	3223584.1	2322093.7	2167877.9
Cycles Desprotegido	1676531.8	1952286.5	1534481.3	1570397.5	1736650.5	2285505.9	1575000.2	1557004.9
Instructions Overhead (%)	45.65	42.35	44.89	49.71	121.77	52.86	61.45	46.04
Instructions Protegido	1415895.6	1686995.3	1295859.0	1361827.5	2342002.9	2113540.1	1515836.3	1325393.8
Instructions Desprotegido	972131.34	1185072.4	894366.32	909615.39	1056069.1	1382637.4	938865.33	907537.06

Otimização	-O3
Programa	yes
Cache-Misses Overhead (%)	12.75
Cache-Misses Protegido	4429.03
Cache-Misses Desprotegido	3928.32
Cache-References Overhead (%)	20.77
Cache-References Protegido	30520.88
Cache-References Desprotegido	25270.96
Branch-Misses Overhead (%)	37.52
Branch-Misses Protegido	12783.26
Branch-Misses Desprotegido	9295.75
Branches Overhead (%)	43.01
Branches Protegido	301812.57
Branches Desprotegido	211047.36
Stalled-Cycles-Frontend Overhead (%)	31.86
Stalled-Cycles-Frontend Protegido	1845761.5
Stalled-Cycles-Frontend Desprotegido	1399794.4
Task-Clock Overhead (%)	31.51
Task-Clock Protegido	0.96
Task-Clock Desprotegido	0.73
Cycles Overhead (%)	36.14
Cycles Protegido	2656431.0
Cycles Desprotegido	1951256.6
Instructions Overhead (%)	44.36
Instructions Protegido	1670641.7
Instructions Desprotegido	1157306.8