МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский ядерный университет «МИФИ»
Кафедра №42 «Криптология и кибербезопасность»

УДК: 004.056.5

Отчет о научно-исследовательской работе
по этапу №1

по теме:

РАЗРАБОТКА ИНСТРУМЕНТА ДЛЯ МОНИТОРИНГА ДОСТУПА К ПАМЯТИ В ЗАДАННОМ ПРОГРАММНОМ КОНТЕКСТЕ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Студент:
Группа Б23-505 В. А. Доценко

Научный руководитель: И. Ю. Коркин

Москва 2026

РЕФЕРАТ

Отчёт 68 с., 7 рис., 4 табл., 53 ист.

МОНИТОРИНГ ПАМЯТИ, LINUX, ДОСТУП К ПАМЯТИ, ЦЕЛОСТНОСТЬ ДАННЫХ, ИНСТРУМЕНТЫ ПРОФИЛИРОВАНИЯ, КОНТЕКСТНЫЙ МОНИТОРИНГ, ЗАЩИТА ПАМЯТИ

Цель работы: создание методики и программного инструмента для мониторинга операций с памятью в процессе в Linux, который позволит проводить фильтрацию операций в зависимости от контекста, включая поток, модуль, системный вызов и адресный диапазон. Задачи исследования включают анализ существующих методов мониторинга в Linux, проектирование архитектуры инструмента, разработку его прототипа и тестирование на соответствие техническим требованиям.

Объектом исследования является мониторинг доступа к памяти в операционной системе Linux, включая методы и инструменты для отслеживания операций с памятью. **Предмет исследования** – разработка инструмента, который обеспечит контекстный мониторинг доступа к памяти в Linux с возможностью фильтрации операций по потоку, модулю, системному вызову и адресу.

Методологическая основа:

Системный подход к анализу механизмов мониторинга памяти и угроз безопасности в операционной системе Linux.

Подход “инструмент мониторинга – механизм защиты”: анализ существующих инструментов мониторинга памяти и сопоставление их с требованиями безопасности.

Фокус на конфиденциальности и целостности данных, как ключевых характеристиках, которые должны быть защищены в процессе мониторинга доступа к памяти.

Сравнительный анализ существующих инструментов мониторинга и технологий безопасности для выявления их преимуществ и недостатков.

Подход к разработке нового инструмента, ориентированный на интеграцию с существующими технологиями Linux и минимизацию воздействия на производительность системы.

Методы исследования:

Сравнительный анализ существующих инструментов мониторинга памяти в Linux.

Моделирование работы системы мониторинга.

Сравнительный анализ методов мониторинга памяти с точки зрения производительности и безопасности: изучение аппаратных технологий и их применимость в контексте инструмента.

Разработка и тестирование нового инструмента мониторинга: создание прототипа инструмента, который будет анализировать доступ к памяти в заданном контексте, с минимальным оверхедом и высокой точностью.

Оценка эффективности инструмента с использованием взвешенной многокритериальной оценки: анализ производительности, точности мониторинга, удобства использования и влияния на безопасность системы.

Ожидается, что предложенный инструмент позволит повысить точность мониторинга, обеспечив фильтрацию данных по контексту, что сделает процесс более гибким и менее ресурсоемким, чем существующие решения. Методология исследования включает проектирование системы, разработку прототипа и экспериментальное тестирование.

Научная новизна заключается в разработке нового подхода к контекстному мониторингу памяти, что расширяет возможности существующих инструментов и вносит вклад в сферу информационной безопасности.

Результаты работы для этапа №1:

1) аналитический обзор и классификация задач мониторинга памяти;
2) сравнительная таблица инструментов по критериям;
3) требования к новому инструменту;
4) предварительная концепция архитектуры.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчёте о НИР применяются следующие термины с соответствующими определениями:

Атрибуция события – привязка события доступа к элементам исполнения программы и структуры адресного пространства (поток, модуль, функция/ инструкция при наличии информации сопоставления адресов).

Аппаратные watchpoints (data breakpoints) – механизм отладочного контура процессора, позволяющий прерывать выполнение программы при доступе (чтении, записи или исполнении) к заданным адресам памяти. Обладает высокой точностью, но крайне ограниченным числом слотов.

Виртуальная память – механизм ОС, предоставляющий процессам абстрактное адресное пространство, отображаемое ядром на физические страницы RAM и (при необходимости) на подкачку.

Виртуальное адресное пространство процесса – диапазон виртуальных адресов, доступных процессу, внутри которого размещаются стек, куча, отображения файлов и другие регионы.

Виртуальный адрес – адрес в виртуальном адресном пространстве процесса, используемый программой при обращениях к памяти.

Динамическая бинарная инструментализация (DBI) – метод анализа программы, при котором её исполняемый код модифицируется во время выполнения для вставки проверок или сбора данных.

Использование освобождённой памяти (Use-After-Free) – обращение к памяти после её освобождения, что может привести к непредсказуемым последствиям, таким как использование устаревших или перезаписанных данных.

Контекст выполнения – совокупность признаков, связывающих событие доступа с тем, кто и при каких условиях его совершил (процесс/поток, модуль, точка выполнения, системный контекст, диапазон адресов).

Контекстная фильтрация – отбор подмножества событий доступа по заданным условиям (тип R/W/X, диапазоны адресов, потоки, модули, системный контекст) для снижения объёма данных и повышения интерпретируемости.

Memory safety уязвимости – класс дефектов управления памятью, позволяющих нарушать границы объектов, обращаться к освобождённой

памяти или некорректно использовать указатели, что может приводить к компрометации.

Переполнение буфера (buffer overflow) – запись или чтение за границы буфера, приводящие к повреждению соседних данных или управляющих структур.

Резидентный набор (RSS) – часть памяти процесса, которая в данный момент находится в RAM (в физической памяти), а не вытеснена.

Событие доступа к памяти – фиксируемый факт чтения, записи или исполнения по адресу (или диапазону) в виртуальном адресном пространстве процесса с набором атрибутов (адрес, тип, размер, время, контекст).

Тип доступа (R/W/X) – классификация доступа к памяти как чтение (R), запись (W) или исполнение (X).

Системный вызов (syscall) – интерфейс перехода из пользовательского режима в ядро; в мониторинге используется как маркер системного контекста, особенно для операций, меняющих карту памяти (например, mmap, mprotect).

Страница памяти (page) – фиксированный по размеру блок виртуальной и физической памяти, являющийся базовой единицей отображения и защиты в ОС.

Отображение (mapping) – сопоставление диапазона виртуальных адресов процесса с источником данных (анонимная память или файл) и с набором прав доступа.

EPT (Extended Page Tables) – это аппаратная технология, которая является частью архитектуры процессоров, поддерживающих виртуализацию.

Uprobes / Kprobes – механизмы ядра Linux для установки динамических точек трассировки (пробов) в пользовательском (uprobes) или kernel (kprobes) пространстве

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете о НИР применяют следующие сокращения и обозначения:

ASan – AddressSanitizer, инструмент для обнаружения ошибок работы с памятью (memory bugs) на основе компиляторной инструментализации.

ASLR – Address Space Layout Randomization, рандомизация адресного пространства.

CC – Confidential Computing, парадигма защиты обрабатываемых данных в изолированных доверенных средах выполнения (TEE).

DAMON (Data Access MONitor) – подсистема ядра Linux для оценки паттернов доступа к памяти на уровне диапазонов, предназначенная для выявления «горячих» областей с целью адаптивного повышения детализации мониторинга.

DBI – Dynamic Binary Instrumentation, динамическая бинарная инструментализация.

eBPF – Extended Berkeley Packet Filter, механизм фильтрации и мониторинга событий в ядре ОС Linux.

Intel PT – Intel Processor Trace, аппаратная технология для эффективной записи трассы выполнения программы.

MMU – Memory Management Unit, блок управления памятью.

NX – No-eXecute, запрет исполнения со страниц.

PAGEFAULT – ошибка страницы, связанная с нарушением прав доступа.

PAGEMAP – интерфейс ядра Linux (/proc/pid/pagemap), предоставляющий доступ к информации о состоянии физических страниц, отображенных в адресное пространство процесса.

PID – Process ID, идентификатор процесса.

PXN – Privileged Execute Never, атрибут страницы в архитектуре ARM, запрещающий исполнение кода из этой страницы в привилегированном режиме.

R/W/X – Read/Write/eXecute, типы доступа.

RSS – Resident Set Size, резидентный набор.

Soft-dirty – механизм отслеживания изменений в страницах памяти в Linux.

TEE – Trusted Execution Environment, изолированная безопасная область выполнения процессора, используемая в Confidential Computing.

TID – Thread ID, идентификатор потока.

UAF – Use-After-Free, использование освобождённой памяти.

ВВЕДЕНИЕ

Мониторинг доступа к памяти процесса – ключевая задача в области информационной безопасности и разработки программного обеспечения. Эффективное отслеживание операций чтения, записи и исполнения данных в памяти играет важную роль в таких областях, как отладка программ, реверс-инжиниринг, анализ вредоносного ПО, защита от эксплойтов, а также обеспечение целостности данных и выявление внедренного кода. В условиях растущей сложности программ и новых угроз, возникающих из-за уязвимостей, задача мониторинга доступа к памяти становится всё более актуальной.

Актуальность данного исследования обусловлена значительным запросом со стороны государства и растущими потребностями индустрии. Государственная политика в сфере технологического суверенитета и безопасности информационной инфраструктуры, отражённая в Указе Президента РФ № 400 [1], определяет развитие защищённых отечественных технологий как стратегический приоритет. Доктрина информационной безопасности [2] прямо указывает на необходимость создания систем обнаружения и противодействия киберугрозам, включая инструменты низкоуровневого мониторинга. Правовую основу для таких разработок формирует Федеральный закон № 149-ФЗ [3], устанавливающий требования к защите информации от несанкционированного доступа.

Параллельно индустрия демонстрирует устойчивый рост спроса на решения для анализа выполнения кода в реальном времени. Рынок инструментов мониторинга прогнозирует увеличение в сегменте безопасности [4], что связано с распространением атак, направленных на эксплуатацию уязвимостей в памяти. Современные вредоносные программы активно используют техники обфускации и внедрения в память, что делает их недоступными для классических сигнатурных сканеров. Как показывают исследования, анализ содержимого и операций с памятью существенно повышает точность обнаружения скрытых угроз [5,6]. Кроме того, развитие парадигмы конфиденциальных вычислений (Confidential Computing) создаёт новый класс задач: для верификации безопасности изолированных сред выполнения (таких как Intel SGX или AMD SEV) требуются специализированные инструменты, способные контролировать доступ к защищённым областям памяти без полного доверия к операционной системе

[7]. Это формирует потребность в эффективных механизмах мониторинга, сочетающих низкие накладные расходы с возможностью контекстно-зависимой фильтрации событий.

Современные инструменты мониторинга, такие как strace, gdb и другие профайлеры, предоставляют базовые функции отслеживания операций с памятью, однако они не поддерживают фильтрацию по контексту работы программы, что ограничивает их применимость в сложных многозадачных приложениях. Более продвинутые решения, такие как MemGaze, предлагают низкоуровневый анализ трасс доступа к памяти с использованием аппаратной поддержки, что позволяет получать высокое разрешение анализа, но ориентированы больше на анализ шаблонов доступа, чем на контекстную фильтрацию в реальном времени [8]. Аналогично, подходы наподобие AutoProfile автоматизируют генерацию профилей для анализа памяти, однако их возможности для динамического контекстного мониторинга ограничены [9]. Другие исследования, такие как MTM, переосмысливают методы профилирования памяти для многоуровневых систем памяти, демонстрируя различные возможности и ограничения современных методов [10]. Проекты типа MemLiner ориентированы на оптимизацию взаимодействия трассировки и выполнения приложения в средах с удаленной памятью [11]. Исследования в области аппаратной поддержки, такие как MemTracker [12], предлагают эффективные программируемые механизмы для мониторинга и отладки доступа к памяти, однако требуют модификации аппаратной архитектуры и в основном сфокусированы на отладке ошибок, а не на задачах контекстного наблюдения в реальном времени. Мониторинг через обработку page faults с использованием механизмов вроде `userfaultfd` [13] также рассматривается, но страдает от высоких накладных расходов при частых обращениях и сложности реализации контекстной фильтрации.

Анализ существующих исследований и инструментов выявляет ряд системных недостатков. Большинство решений либо обеспечивают низкоуровневый мониторинг с высокими накладными расходами и без поддержки семантического контекста (MemGaze, подходы с `userfaultfd`), либо фокусируются на узких задачах, таких как автоматическое профилирование (AutoProfile), отладка (MemTracker) или оптимизация для специфических архитектур (MTM, MemLiner). Ключевым нерешённым вопросом остаётся создание инструмента, который бы сочетал эффективный мониторинг операций с памятью в реальном времени с возможностью

интеллектуальной, контекстно-зависимой фильтрации событий. Именно этот пробел определяет научную проблему, на решение которой направлено настоящее исследование.

1 Постановка задачи

Целью всей работы является разработать методику и программный инструмент для мониторинга операций чтения/записи/исполнения в память целевого процесса в Linux, с возможностью фильтрации по контексту (поток, модуль, системный вызов, адресный диапазон).

Основные задачи всей работы:

1) провести сравнительный анализ существующих подходов к мониторингу доступа к памяти в Linux;
2) спроектировать архитектуру инструмента для контекстного мониторинга доступа к памяти в Linux;
3) разработать и реализовать прототип инструмента для мониторинга доступа к памяти в заданном контексте в Linux;
4) провести тестирование разработанного прототипа, которое позволит всесторонне оценить работоспособность ключевых функций и убедиться в соответствии продукта техническим требованиям.

В рамках этапа №1 поставлены следующие задачи:

1) анализ задач мониторинга памяти. Провести обзор существующих подходов и инструментов мониторинга памяти (в том числе профилировщики, системные метрики, встроенные средства ОС и фреймворков);
2) сравнительный анализ существующих инструментов, механизмов ОС, и аппаратных технологий. Выполнить системное сравнение существующих инструментов мониторинга памяти, встроенных механизмов операционных систем и аппаратных технологий с целью выявления их возможностей, ограничений и применимости в различных сценариях использования;
3) формирование требований к инструменту. Необходимо выявить и систематизировать требования к разрабатываемому инструменту. В рамках работы требуется определить функциональные, эксплуатационные и пользовательские требования, включая ограничения и показатели качества;
4) проектирование архитектуры на концептуальном уровне. В рамках работы необходимо определить ключевые компоненты, а также описать общую структуру взаимодействия между ними.

В результате выполнения научно-исследовательской работы предполагается получение следующих результатов:

1) аналитический обзор и классификация задач мониторинга памяти;
2) сравнительная таблица инструментов по критериям;
3) требования к новому инструменту;
4) предварительная концепция архитектуры.

Ключевая решаемая научная проблема в рамках данной работы заключается в противоречии между высоким уровнем угроз, связанных с уязвимостями в механизмах мониторинга памяти в операционной системе Linux, и недостаточной проработанностью подходов к эффективному мониторингу доступа к памяти в специфических контекстах (например, по потокам и адресным диапазонам). С одной стороны, современные системы безопасности и отладки активно используют инструменты для мониторинга памяти, с другой — существующие методы мониторинга имеют ограничения в контекстной фильтрации и не обеспечивают полного контроля за операциями памяти. Эти подходы разрознены и не предлагают единой платформы для анализа угроз и разработки защитных механизмов, что затрудняет их внедрение и объективную оценку.

Сохранение указанного противоречия может привести к снижению безопасности систем, использующих Linux, и увеличению рисков эксплуатации уязвимостей. Возможное разрешение данной проблемы заключается в разработке унифицированного инструмента для мониторинга доступа к памяти, с возможностью фильтрации по контексту, что позволит повысить точность анализа, улучшить защиту и минимизировать угрозы, связанные с несанкционированным доступом к памяти.

2 Анализ задач мониторинга памяти

Современные прикладные системы одновременно растут по сложности и по требованиям к надёжности, из-за чего ошибки работы с памятью всё чаще проявляются не как единичные дефекты, а как системный источник деградации и уязвимостей. Оперативная память и механизмы виртуальной памяти связывают воедино вычисления, ввод-вывод и безопасность, поэтому наблюдаемость поведения памяти становится отдельной инженерной задачей. В реальных системах важно уметь отличать нормальные паттерны обращений к памяти от аномальных, а также связывать изменения поведения с конкретными компонентами и фазами выполнения. При этом мониторинг памяти не сводится к одной метрике: на практике различают наблюдение состояния памяти как ресурса и наблюдение событий доступа к памяти на уровне процессов и потоков.

2.1 Введение в задачи мониторинга памяти

Перед формулированием конкретных задач полезно зафиксировать, что в Linux термин «память процесса» описывает не физические адреса RAM, а виртуальное адресное пространство, отображаемое ядром на физические страницы. Это означает, что наблюдаемые адреса и диапазоны в первую очередь имеют смысл в рамках процесса и его отображений, а не как «позиции» в оперативной памяти.

На уровне анализа поведения приложений центральной единицей становится событие доступа с атрибутами адреса, типа операции и контекста выполнения. Контекст позволяет связывать события с потоком, модулем и фазой работы программы и тем самым повышает интерпретируемость результатов. В практических сценариях без контекста мониторинг быстро превращается в набор разрозненных адресов, который сложно сопоставить с логикой приложения. В дальнейшем под мониторингом памяти преимущественно понимается мониторинг доступов к памяти целевого процесса в Linux с возможностью контекстной фильтрации.

2.1.1 Основные понятия мониторинга памяти

Оперативная память (RAM – Random Access Memory) представляет собой основной высокоскоростной ресурс хранения данных во время

выполнения программ. В Linux доступ приложений к RAM опосредован механизмом виртуальной памяти: каждый процесс имеет виртуальное адресное пространство, разделённое на диапазоны (отображения), которые ядро отображает на физические страницы.

Для описания поведения процесса на уровне памяти удобно использовать понятия «виртуальный адрес», «диапазон виртуальных адресов» и «отображение» (mapping), то есть связанный диапазон адресов с атрибутами прав доступа и происхождения данных (анонимная память или отображение файла). Чтобы визуально закрепить устройство адресного пространства процесса и связь его областей с описателями памяти ядра Linux, далее приведена упрощённая схема, отражающая типичные сегменты (text, data, bss), динамические области (heap и mmap) и стек, а также роль структуры mm_struct как описателя адресного пространства.

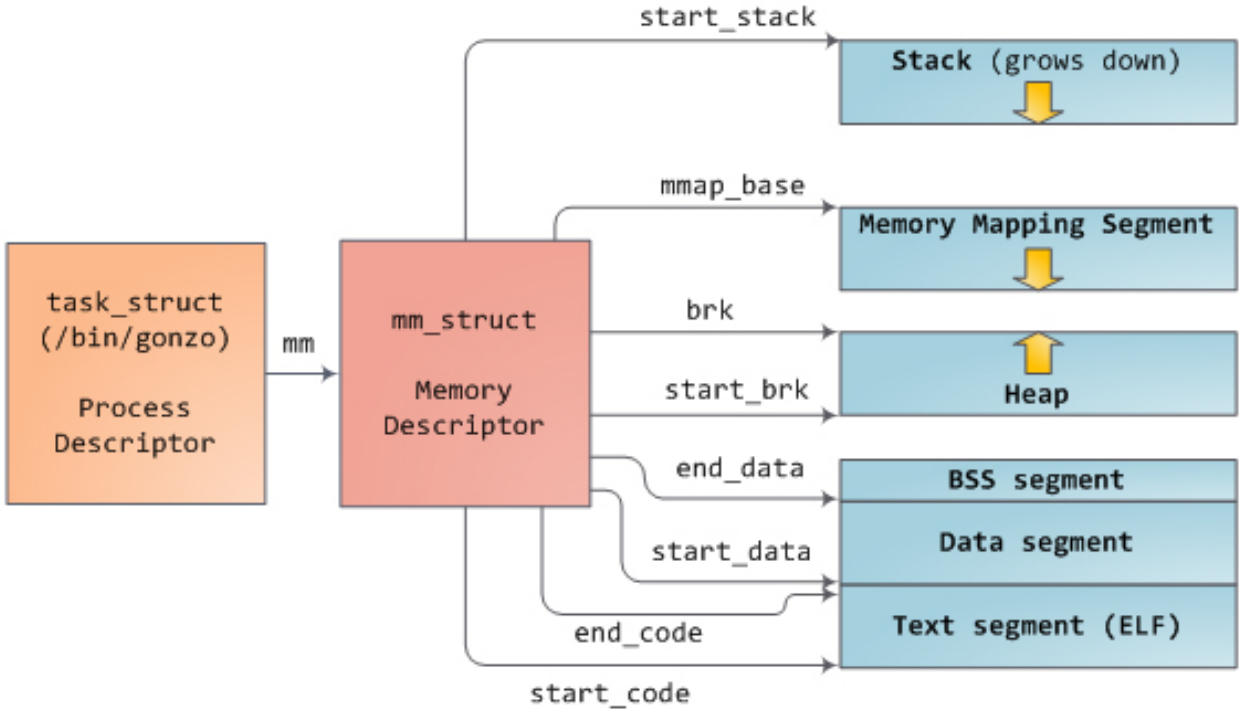

Рисунок 1 — Упрощённая схема виртуального адресного пространства процесса Linux

Схема носит иллюстративный характер и показывает типичную композицию областей процесса, однако конкретные адреса и взаимное расположение регионов зависят от архитектуры, настроек системы и механизмов рандомизации (например, ASLR – Address Space Layout Randomization). Тем не менее она полезна для постановки задач мониторинга, потому что подчёркивает, что наблюдение ведётся по виртуальным адресам процесса, а смысл событий зависит от того, в какой области произошёл доступ. Например, обращения к стеку и куче часто связаны с ошибками управления

памятью, а обращения к отображениям (mmap) отражают как загрузку модулей и библиотек, так и работу с file-backed памятью. На практике мониторинг событий доступа интерпретируется через принадлежность адресов к таким областям и через их права (чтение, запись, исполнение), что позволяет локализовать аномалии и повышать интерпретируемость результатов.

Мониторинг памяти в инженерной практике включает два взаимодополняющих направления. Первое направление – мониторинг состояния памяти как ресурса: выявление утечек, роста потребления, подкачки, давления на память и аномалий распределения памяти между компонентами. Второе направление – мониторинг доступов к памяти: фиксация операций чтения (read), записи (write) и исполнения (execute) в адресном пространстве процесса с возможностью атрибуции событий к контексту выполнения. Для задач данной НИР ключевым является второе направление, поскольку оно позволяет связывать аномальные обращения к памяти с конкретными потоками и программными компонентами.

2.1.2 Роль мониторинга памяти в современных вычислительных системах

Современные вычислительные системы состоят из центрального процессора, оперативной памяти и подсистемы ввода-вывода, причём общая производительность приложений во многом определяется скоростью доступа к данным. Оперативная память выполняет роль хранилища данных и промежуточного буфера, обеспечивающего выполнение программ и обмен с внешними устройствами. При росте объёмов данных и усложнении программ ограничения подсистемы памяти становятся одним из главных факторов, сдерживающих производительность вычислений, что в литературе связывают с явлением *memory bottleneck*.

Исследования архитектуры вычислительных систем подчёркивают, что задержки и пропускная способность памяти являются критическими ограничителями эффективности вычислений, особенно при масштабировании нагрузки и росте параллелизма [14]. По мере роста требований приложений к обработке данных ограничения проявляются сильнее: растёт значение пропускной способности, латентности и эффективности использования интерфейсов памяти [15]. В такой ситуации наблюдение поведения памяти позволяет обнаруживать «узкие места», которые могут оставаться

незаметными при анализе только процессорной нагрузки или суммарного потребления ресурсов [9]. Практическая ценность мониторинга здесь заключается в том, что он позволяет переходить от общих симптомов (деградация, рост потребления) к локализуемым причинам (конкретные диапазоны, потоки и фазы выполнения).

2.1.3 Мониторинг памяти как инструмент безопасности

Подсистема памяти является критической областью не только для производительности, но и для информационной безопасности. Ошибки управления памятью, включая выход за пределы буфера, обращение к освобождённой памяти и некорректные операции с указателями, приводят к нарушению целостности данных, аварийному завершению программ и создают основу для эксплуатации уязвимостей [16]. Такие ошибки лежат в основе широкого класса атак, где злоумышленник стремится либо изменить поток исполнения программы, либо получить доступ к чувствительным данным процесса.

Классическим примером является переполнение буфера, позволяющее повлиять на управляющие структуры и вызвать отклонение потока исполнения; развитие этой идеи включает атаки повторного использования кода, такие как ROP (Return-Oriented Programming), которые комбинируют фрагменты уже существующего кода для обхода ряда защитных механизмов. С практической точки зрения мониторинг поведения памяти важен тем, что позволяет фиксировать аномальные паттерны доступа (неестественные записи в чувствительные диапазоны, нетипичные траектории исполнения, повторяемые обращения к определённым регионам), которые могут выступать индикаторами эксплуатации уязвимостей. По сравнению со статическим анализом мониторинг в реальном времени отражает фактическое поведение программы в условиях эксплуатации и тем самым помогает точнее оценивать угрозы и последствия компрометации [17].

2.2 Основные сценарии использования

В рамках анализа задач мониторинга памяти можно выделить несколько ключевых сценариев, которые направлены на защиту данных и обеспечение безопасности приложений и операционной системы. Рассмотрим два основных из них.

2.2.1 Защита обрабатываемой информации в памяти

Одним из важнейших сценариев использования инструмента является защита конфиденциальных данных, которые обрабатываются в памяти процесса. [18] В современных приложениях часто содержится информация, которая требует защиты, например, ключи шифрования, пароли, персональная информация и другие чувствительные данные.

Мониторинг доступа к этим данным в памяти помогает выявить несанкционированные операции с конфиденциальной информацией. Инструмент для мониторинга памяти фиксирует все операции с памятью, включая чтение и запись в области, где могут храниться такие данные. В случае выявления доступа к защищенным данным инструмент записывает это событие в отчет, указывая:

– процесс, который осуществил доступ,

– тип операции,

– адрес памяти, к которому был произведен доступ.

Это позволяет позже провести анализ событий, выявив возможные попытки несанкционированного доступа к конфиденциальной информации. Этот сценарий особенно полезен для отслеживания атак, таких как утечка данных, инъекции кода или другие попытки получения доступа к защищенным данным.

2.2.2 Контроль целостности глобальных флагов в памяти

Другим важным направлением является контроль целостности критических данных, таких как глобальные флаги аутентификации, указатели и другие внутренние переменные, которые играют ключевую роль в логике работы программ [18]. Нарушение целостности этих данных может привести к атакам, направленным на манипулирование внутренним состоянием приложения или операционной системы.

Инструмент фиксирует все изменения в таких данных, записывая в отчет каждую операцию с соответствующими флагами и указателями. Как и в прошлом сценарии, инструмент фиксирует основные параметры активности (тип операции, процесс, адрес памяти) в отчете. Такой мониторинг позволяет создать полный отчет о целостности критических данных и служит основой для дальнейшего анализа на предмет возможных атак, например,

изменения флагов аутентификации или попыток манипулирования внутренним состоянием программы.

2.3 Основные задачи

2.3.1 Наблюдение операций чтения, записи и исполнения в память

Мониторинг доступа к памяти на уровне процесса включает наблюдение трёх фундаментальных типов операций: чтение (read), запись (write) и исполнение (execute) в адресном пространстве процесса. Чтение описывает получение данных из памяти в регистры или внутренние буферы процессора, запись – изменение содержимого памяти, а исполнение – выборку и выполнение инструкций, расположенных по адресам в виртуальном адресном пространстве. В терминах мониторинга каждая операция характеризуется типом доступа, адресом (или диапазоном адресов), объёмом затронутых данных и моментом времени.

На практике события доступа интерпретируются не по одиночке, а как последовательности и паттерны: повторяемые записи в один диапазон могут указывать на неэффективные паттерны обновления данных или на активность, изменяющую критичные структуры; частые обращения к одним и тем же данным могут отражать особенности локальности и поведения алгоритма [19]. С точки зрения безопасности особый интерес представляют случаи, когда характер исполнений или записей в память становится нетипичным для нормального профиля программы, что может сопровождать эксплуатацию уязвимостей управления памятью, включая переполнения буфера [20]. Для корректного анализа важно помнить, что в современных системах исполнение произвольных данных часто ограничено политиками прав страниц; поэтому наблюдение исполнения используется как индикатор изменений поведения исполнения и привязки к диапазонам, а не как предположение, что «данные всегда исполняются».

2.3.2 Определение наблюдаемого события доступа к памяти

Для согласованного мониторинга необходимо формально определить параметры наблюдаемого события доступа к памяти. Ниже перечислены параметры, которые в большинстве сценариев являются минимально достаточными для анализа производительности и безопасности:

1) **адрес доступа** – виртуальный адрес в адресном пространстве процесса, либо границы диапазона, если событие агрегируется по региону;
2) **тип доступа** – чтение, запись или исполнение, то есть принадлежность события к классу R/W/X;
3) **размер доступа** – объём данных, затронутых событием (для анализа интенсивности и для различения, например, точечных и блочных операций);
4) **временная характеристика** – временная метка или порядок события, позволяющие анализировать частоту, повторяемость и причинно-следственные связи между событиями;
5) **контекст выполнения** – идентификатор процесса и потока, а также дополнительная информация, позволяющая привязать событие к коду (например, модуль выполнения) и к управляющим действиям ОС (например, системный вызов, изменивший карту памяти).

Указание физического адреса RAM для задач мониторинга доступа к памяти в пользовательском адресном пространстве обычно не требуется, поскольку анализ строится в терминах виртуальных адресов процесса и его отображений.

2.3.3 Атрибуция доступа к памяти и контекстная фильтрация

Атрибуция доступа к памяти – это процесс привязки события доступа к конкретному потоку, инструкции, функции, модулю, системному вызову или адресу в виртуальном адресном пространстве. Этот процесс позволяет точно идентифицировать, какой элемент программы вызывает операцию с памятью и в каком контексте. В результате, атрибуция помогает локализовать проблему, будь то ошибка в программе или уязвимость безопасности, и отслеживать её происхождение, что критически важно для диагностики производительности или безопасности системы. Виды атрибутов:

1) **поток** – мониторинг на уровне потоков позволяет определить, какой из множества параллельных потоков выполняет операции с памятью. В многозадачных и многопроцессорных системах каждый поток может работать с разными участками памяти, что делает важным анализ на уровне потоков для мониторинга параллельных вычислений и выявления узких мест;

2) **модуль** – привязка события к исполняемому файлу или разделяемой библиотеке помогает связывать обращения к памяти с конкретными компонентами и зависимостями;
3) **функция или инструкция** – более точная привязка к месту в коде позволяет локализовать проблемный фрагмент, однако на практике степень точности зависит от доступности информации сопоставления адресов к коду (например, наличия отладочной информации DWARF [21] и таблиц символов ELF [22]);
4) **системный вызов** – системные вызовы не «выполняют» чтения и записи пользовательского кода напрямую, но часто задают контекст изменения карты памяти и буферов (выделение отображений, смена прав, операции ввода-вывода), поэтому полезно связывать события доступа с близкими по времени управляющими действиями ОС [23];
5) **диапазон в виртуальном адресном пространстве** – отнесение адреса к стеку, куче, анонимному отображению или отображению файла облегчает интерпретацию событий и помогает выделять чувствительные зоны для анализа.

Контекстная фильтрация – это выделение подмножества событий, которые имеют значение для конкретной задачи анализа. В отличие от атрибуции, которая дополняет событие метаданными о том, кто и где в программе инициировал доступ, контекстная фильтрация использует уже полученные атрибуты как условия отбора и тем самым уменьшает объём наблюдений и повышает отношение сигнал/шум.

Фильтрация позволяет ограничивать объём наблюдений, повышать отношение сигнал/шум и делать мониторинг применимым в реальных сценариях. Типичные критерии фильтрации включают тип доступа (R/W/X), выбранные диапазоны виртуальных адресов, конкретные потоки или модули, а также ограничение наблюдений событиями, происходящими в заданном системном контексте (например, после выделения нового отображения или смены прав страниц).

2.4 Трудности и ограничения

2.4.1 Накладные расходы

Одной из основных проблем при мониторинге доступа к памяти является значительная накладная нагрузка, которую он накладывает на систему. Это связано с тем, что для мониторинга всех операций с памятью требуется постоянный контроль над огромным количеством данных, что может существенно снизить производительность системы. Как показывают исследования [14], полная трассировка операций доступа к памяти может быть практически непрактична, особенно на системах с высокими требованиями к скорости обработки данных и с большим числом одновременно выполняющихся процессов и потоков.

Основная сложность заключается в том, что мониторинг с максимальной детализацией требует больших вычислительных ресурсов. Для снижения накладных расходов разработаны различные методы выборочного мониторинга, которые могут ограничивать сбор данных до наиболее значимых операций. Например, использование специализированных фильтров, таких как eBPF (Extended Berkeley Packet Filter [24,25]), позволяет значительно уменьшить накладные расходы, не снижая при этом качества мониторинга. Тем не менее, даже такие методы могут быть неэффективными на крупных системах с большим объёмом данных, что ограничивает их применение в реальных условиях. Это также увеличивает сложность работы с данными, что делает анализ менее точным и более трудозатратным.

2.4.2 Масштабируемость

Масштабируемость – это способность системы мониторинга эффективно работать с увеличивающимся объёмом данных при росте числа потоков, процессов или объёмов адресного пространства. По мере роста системы и количества взаимодействующих компонентов мониторинг должен оставаться эффективным, а его стоимость не должна пропорционально увеличиваться. Важно, чтобы мониторинг мог масштабироваться без значительного увеличения вычислительных затрат и обеспечивал работу в условиях больших объёмов данных.

Сложность масштабирования также заключается в поддержке мониторинга с сохранением точности в условиях параллельных вычислений,

многозадачности и использования динамически изменяющихся виртуальных адресных пространств. Обработка данных должна происходить быстро и эффективно, а инструмент должен быть готов к увеличению числа потоков и процессов без значительного увеличения нагрузки на систему.

2.4.3 Неполнота наблюдений

Неполнота наблюдений – это одна из значимых проблем, с которой сталкиваются системы мониторинга памяти. В условиях ограниченных ресурсов или при выборочном мониторинге часто возникает ситуация, когда некоторые события доступа к памяти пропускаются, что приводит к недооценке объёмов памяти, которые могут быть использованы неправильно или привести к проблемам с безопасностью.

Исследования показали, что высокая нагрузка на систему, особенно в многозадачных средах, значительно увеличивает вероятность пропуска данных при мониторинге. Например, системы динамического профилирования памяти часто сталкиваются с трудностью точного отслеживания всех операций из-за ограничений по частоте выборки и обработке больших объёмов данных [15]. В многопоточных системах дополнительные проблемы возникают при анализе операций, связанных с параллельными потоками и процессами, что ещё больше усложняет задачу обеспечения полноты наблюдений. Это, в свою очередь, ведёт к неточности в анализе поведения программы и повышает вероятность пропусков критичных событий.

2.5 Выводы

В данной главе рассматривались ключевые задачи мониторинга памяти в современных вычислительных системах, с акцентом на их роль в обеспечении безопасности и стабильности программ. Были рассмотрены основные аспекты мониторинга, включая отслеживание операций чтения, записи и исполнения в память, а также необходимость контекстной фильтрации для повышения точности анализа. Также было выделено значительное внимание важности мониторинга как инструмента для защиты данных. Определены основные сценарии использования инструментов мониторинга памяти. Были сформированы основные задачи и ограничения мониторинга доступа к памяти.

Основные задачи мониторинга памяти включают:

– отслеживание операций чтения, записи и исполнения в виртуальном адресном пространстве,
– атрибуцию этих операций к конкретным потокам, модулям, функциям или системным вызовам для более точного анализа,
– использование контекстной фильтрации для уменьшения объёма данных, что помогает сосредоточиться на наиболее значимых событиях.

Основные ограничения, с которыми сталкиваются системы мониторинга памяти, включают:

1) **накладные расходы**, связанные с высокой частотой мониторинга, что может негативно сказываться на производительности системы. Для решения этой проблемы необходимо найти баланс между частотой сбора данных и их детализацией, используя методы выборочного мониторинга, такие как фильтрация несущественных операций;
2) **ограничения в масштабируемости мониторинга**, когда системы должны эффективно обрабатывать увеличивающиеся объёмы данных, что требует интеграции лёгких технологий мониторинга, способных работать с минимальными затратами на сбор и обработку данных;
3) **проблемы с полнотой наблюдений**, которые могут возникать при высоких нагрузках на систему, что ведёт к пропуску событий. Для минимизации этой проблемы инструмент мониторинга должен включать возможность агрегации данных по диапазонам виртуальных адресов или времени, что позволяет снизить объём данных и сосредоточиться на наиболее значимых событиях.

Проектирование инструмента мониторинга должно учитывать эти ограничения и направлено на создание эффективной системы, которая будет обеспечивать высокий уровень детализации при минимальных накладных расходах и высокой производительности, что критично для успешного применения мониторинга в реальных условиях эксплуатации.

3 Сравнительный анализ существующих инструментов, механизмов ОС, и аппаратных технологий

Глава посвящена сравнительному анализу существующих инструментов и технологий, используемых для мониторинга памяти в современных вычислительных системах. В ней рассматриваются как аппаратные, так и программные средства, доступные для мониторинга доступа к памяти, с целью их оценки по различным критериям, таким как накладные расходы, охват адресного пространства и области применимости. Проводится анализ технологий, встроенных в ОС Linux, и аппаратных механизмов, с целью определения их сильных и слабых сторон в контексте обеспечения безопасности и производительности. Результаты данного анализа служат основой для последующих рекомендаций по выбору инструментов для мониторинга памяти в различных сценариях использования.

3.1 Аппаратные технологии

3.1.1 Аппаратные watchpoints (data breakpoints) по адресу

Аппаратные watchpoints опираются на отладочный контур процессора, который умеет сравнивать адрес фактического обращения к памяти с заранее заданными “сторожевыми” адресами и реагировать синхронным событием в момент совпадения. На практике это реализуется отдельным debug unit, который хранит несколько слотов с адресами и условиями срабатывания, а затем встраивается в конвейер выполнения так, чтобы проверка происходила именно в момент доступа, а не постфактум. Для семейства x86-64 такая логика описывается через механизм debug registers, где адреса размещаются в DR0–DR3, а условия доступа и длина наблюдаемого диапазона кодируются управляющими полями, связанными с DR7, при этом само событие относится к архитектурным debug exceptions [26,27]. На Arm аналогичный смысл имеют watchpoint resources self-hosted debug, где адрес и маскирование задаются в специализированных регистрах watchpoint value/ control, а обработчик получает диагностическую информацию через регистры синдрома исключения [28].

Событием для watchpoint является аппаратное исключение отладки, возникающее строго в точке обращения к памяти, попавшей под заданные условия. Важная деталь здесь в том, что условия обычно различают тип

операции: запись, чтение или комбинированный режим, а на некоторых архитектурах и режимах возможно и наблюдение исполнения, хотя “исполнение” чаще удобнее ловить через механизмы прав доступа страниц, а не через debug unit. С точки зрения информативности обработчик получает как минимум адрес совпадения и возможность понять, какой тип доступа вызвал событие, а также полный контекст “где именно выполнялся код” через стандартно сохраняемое состояние исключения (адрес текущей инструкции и регистры), что даёт основу для последующей атрибуции к модулю, функции или цепочке вызовов [26,27].

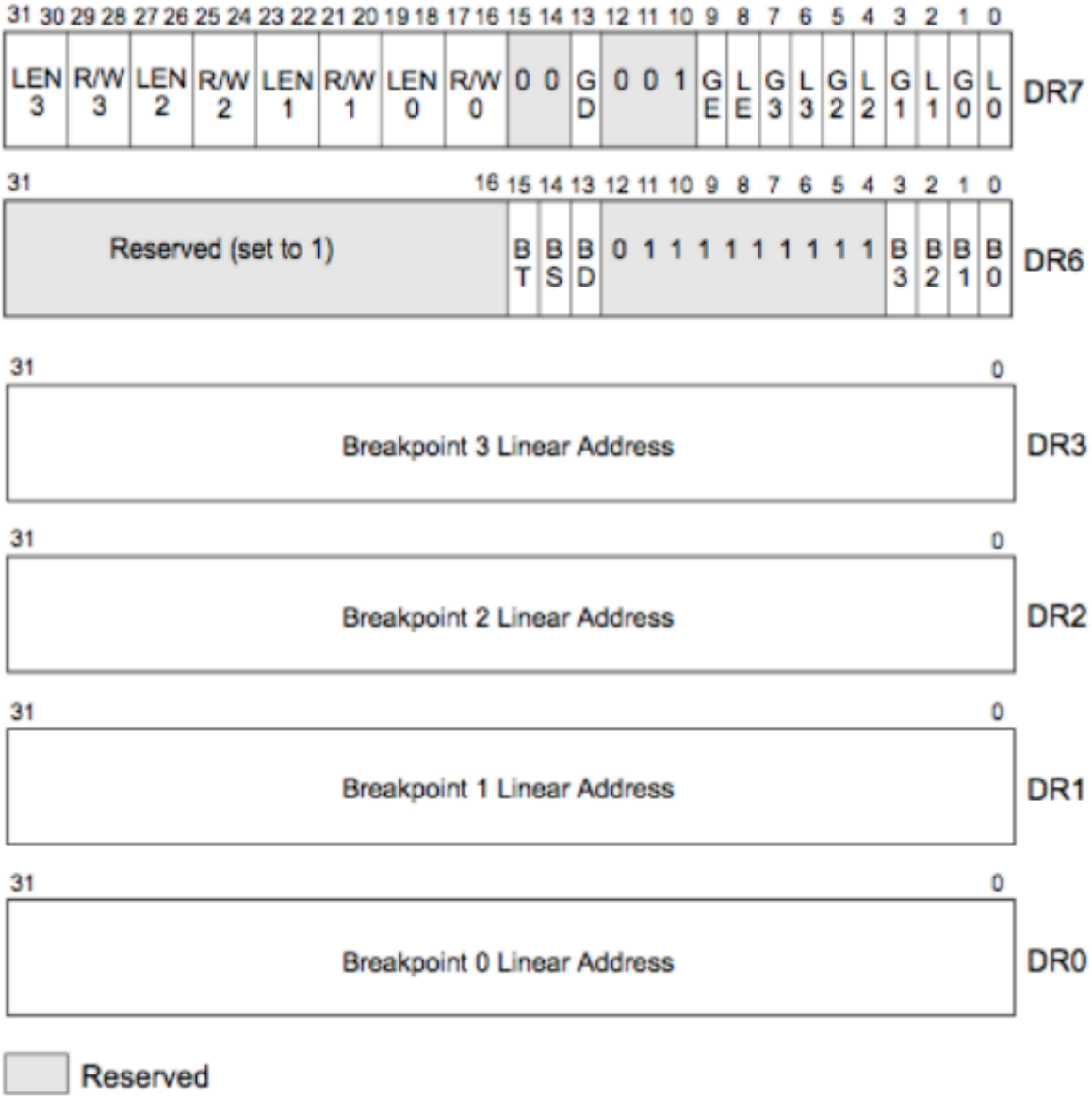

Рисунок 2 — Устройство дебаг-регистров в архитектуре x86

Сильная сторона watchpoints – мелкая гранулярность [29]. В отличие от страничных механизмов, наблюдение можно привязать к конкретному адресу или к небольшому аппаратно задаваемому диапазону вокруг него,

причём диапазон обычно требует выравнивания и задаётся ограниченным набором размеров. Это выглядит как идеальный кандидат для “конкретного доступа к участку памяти”, но дальше быстро проявляется фундаментальное ограничение: число аппаратных слотов очень мало. На x86-64 обычно доступно четыре адресных watchpoint-слота, и даже при аккуратном переиспользовании этого ресурса масштабирование на десятки адресов или на большие диапазоны не получается без динамической перезагрузки конфигурации и дополнительных ухищрений [26,27]. По цене watchpoints удобны, пока срабатывания редкие, потому что само сравнение адресов “дешевле” исключения, но каждое попадание всё равно превращается в синхронный переход управления с сохранением и восстановлением контекста, поэтому наблюдение горячих адресов быстро начинает доминировать в накладных расходах.

Для инструмента мониторинга доступа к памяти в заданном контексте watchpoints хорошо закрывают режимы точечной диагностики, где важна именно точность и момент события. При этом watchpoints почти не подходят на роль универсального механизма для широкого покрытия адресного пространства процесса, и это ограничение лучше проговорить прямо, потому что оно задаётся не реализацией в ОС, а количеством аппаратных ресурсов debug unit.

3.1.2 MMU permission faults: аппаратная проверка прав страниц и исключения

MMU (Memory Management Unit) представляет собой функциональный блок процессора, реализующий виртуальную память на уровне аппаратуры, то есть отображение виртуального адресного пространства процесса на физическую память, а также применение правил доступа, заданных в структурах трансляции. Механизм контроля прав доступа через MMU основан на том, что для большинства современных архитектур обращение к памяти является композицией двух аппаратных процедур: трансляции виртуального адреса и проверки атрибутов доступа. Адресное пространство процесса организуется странично: виртуальная память разбивается на страницы фиксированного размера (типично 4 KiB), а их отображение и свойства задаются записями таблиц страниц, которые содержат как адрес целевого физического фрейма, так и набор атрибутов, определяющих

допустимые операции (чтение, запись, исполнение) и режимы доступа (привилегированность, параметры кэширования и др.) [26].

С аппаратной точки зрения каждая операция загрузки или сохранения (load/store), а также каждая выборка инструкции (instruction fetch), сопровождается обращением MMU к структурам трансляции и проверкой соответствующих атрибутов. Если требуемый тип доступа не разрешён текущими атрибутами страницы, процессор формирует синхронное исключение, фиксируя тем самым нарушение установленной политики доступа. Для x86-64 таким исключением является page fault (PF), при котором faulting linear address сохраняется в CR2 (Control Register 2), а код ошибки (PF error code) содержит диагностические признаки причины нарушения, в частности различение операций записи и, при наличии соответствующей поддержки, нарушений при выборке инструкции [26]. Для Armv8-A аналогичная семантика реализуется через исключения Data Abort и Instruction Abort, где класс и причина нарушения кодируются в ESR_ELx, а адрес, по которому произошёл fault, фиксируется в FAR_ELx.

TLB (Translation Lookaside Buffer) – аппаратный кэш MMU, содержащий результаты недавней трансляции виртуальных адресов и связанные с ними атрибуты доступа, что уменьшает число обращений к таблицам страниц [30].

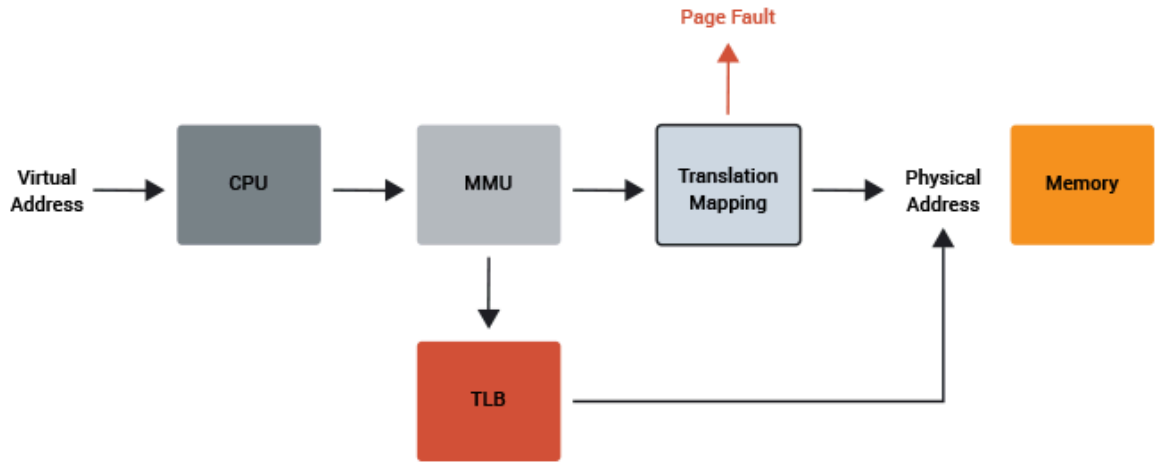

Рисунок 3 — Упрощённая схема трансляции адресов и возникновения page fault

Ключевая особенность permission faults заключается в том, что событие возникает не при любом обращении к памяти, а строго при обращении, которое противоречит текущим правам страницы. Это определяет типичные сценарии применения механизма в задачах мониторинга: он естественным образом подходит для выявления единичных или редких нарушений, а также для

схем “first-touch”, где фиксируется первое обращение к региону памяти после принудительного ограничения прав. Попытка использовать permission faults как источник событий для полного логирования каждого обращения к памяти приводит к необходимости многократного переустановления запретов и тем самым к генерации исключения практически для каждой целевой операции. Поскольку обработка синхронного исключения включает переход управления, сохранение контекста и выполнение обработчика, накладные расходы в таком режиме становятся приблизительно пропорциональными частоте обращений к мониторируемым страницам, что делает мониторинг “каждого доступа” практически непригодным для горячих регионов памяти.

Гранулярность механизма определяется страничной организацией виртуальной памяти. Права доступа задаются на уровне дескрипторов страниц или блоков трансляции, поэтому единицей контроля является страница фиксированного размера или крупная страница (huge page), если такие отображения используются в системе. Применение больших страниц повышает эффективную гранулярность контроля и, как следствие, снижает локальность событий: одно изменение атрибутов может покрывать существенно больший диапазон адресов, чем стандартная страница. Одновременно страничная природа permission faults делает подход удобным в тех случаях, когда интерес представляет мониторинг доступа к целым регионам (например, сегмент данных процесса, область динамических выделений или отображение разделяемой библиотеки), а не к отдельным байтам или объектам.

3.1.2.1 Ловля записи/чтения через запреты на странице

Наиболее прямолинейный вариант применения permission faults в мониторинге заключается в том, что для выбранных страниц устанавливаются ограничения на чтение и/или запись. В таком режиме любая попытка выполнить store в страницу, помеченную как read-only, приводит к синхронному исключению, что аппаратно фиксирует факт записи в защищаемый регион. На x86-64 признак операции записи отражается в поле W/R кода ошибки page fault, а адрес нарушения доступен в CR2, что позволяет однозначно определить виртуальный адрес, спровоцировавший событие [26]. На Armv8-A аналогичный диагностический минимум формируется через

связку ESR_ELx и FAR_ELx, позволяющую различать permission fault и иные причины abort, а также извлечь faulting address.

В момент события мониторинг может опираться на две группы сведений. Первая группа относится к памяти и включает адрес нарушения и тип операции, что позволяет привязать событие к конкретной странице и к классу доступа. Вторая группа относится к контексту исполнения и включает сохранённое состояние потока на момент исключения, в частности адрес инструкции, вызвавшей обращение, и набор регистров. Это создаёт основу для последующей атрибуции события к участку кода и для контекстной фильтрации по признакам исполнения, однако сами по себе аппаратные механизмы не предоставляют высокоуровневых идентификаторов (например, “имя функции” или “имя модуля”), и такие метаданные требуют восстановления на основе адресов и карты отображений процесса.

С точки зрения применимости к инструменту мониторинга запрет записи или чтения на уровне страниц рационально рассматривать как механизм порогового контроля. Он хорошо подходит для фиксации первых обращений к заранее определённым областям или для выявления редких записей в чувствительные регионы, но плохо масштабируется на сценарии, где обращения к выбранным страницам происходят с высокой частотой, поскольку стоимость обработки исключений становится доминирующей, а страничная гранулярность приводит к тому, что события могут агрегировать доступы к множеству различных объектов, размещённых в пределах одной страницы.

3.1.2.2 Ловля исполнения через NX/XN/PXN

Запрет исполнения страниц представляет собой специализированный частный случай permission faults, ориентированный на контроль выборки инструкций. NX (No-eXecute) и XD (eXecute Disable) – обозначения механизма, при котором для страницы памяти задаётся атрибут “неисполняемая”, то есть запрещается выборка инструкций из данного диапазона. В контексте x86-64 принято говорить о бите NX/XD в записи таблицы страниц, а также о поддержке execute-disable на уровне архитектуры, благодаря которой попытка instruction fetch из неисполняемой страницы приводит к синхронному исключению page fault (PF). В этом случае нарушение можно отличить от чтения или записи по признакам кода ошибки PF, а faulting address фиксируется в CR2 [26].

В Armv8-A эквивалентная идея выражается через атрибуты XN (eXecute Never), которые задаются в дескрипторах трансляции и запрещают исполнение из соответствующего блока или страницы. Для более точного разграничения режимов используются UXN (Unprivileged eXecute Never) и PXN (Privileged eXecute Never), позволяющие независимо запрещать исполнение в непривилегированном и привилегированном режимах соответственно. Это означает, что одна и та же область памяти может быть неисполняемой для кода пользовательского уровня и при этом оставаться исполняемой для кода более привилегированных уровней, либо наоборот, в зависимости от выбранной политики защиты [31].

Данный механизм имеет высокую практическую значимость для задач информационной безопасности, поскольку предоставляет аппаратный триггер на попытку исполнения из регионов, которые по политике процесса должны интерпретироваться как данные. В момент события доступны адрес страницы, из которой производилась попытка выборки, и адрес инструкции, вызвавшей обращение, что позволяет реконструировать связь между исполняемым кодом и регионом памяти, нарушившим политику исполняемости [26,32]. Ограничения механизма совпадают с общими ограничениями permission faults: он естественно работает как детектор нарушений или как средство точечного контроля, но не подходит для получения полной трассы исполнения “по памяти”, так как попытка генерировать событие на каждую выборку приводит к неприемлемым накладным расходам из-за частых синхронных исключений.

3.1.3 Extended Page Tables

Extended Page Tables (EPT) – это технология аппаратной виртуализации, которая позволяет ускорить трансляцию адресов в виртуализованных средах. Она добавляет второй уровень трансляции, обеспечивая перевод виртуальных адресов гостевой операционной системы в физические адреса хоста. Это осуществляется с помощью дополнительной таблицы, управляющей правами доступа для каждой страницы как для гостевой ОС, так и для хоста, что способствует изоляции и безопасности виртуализированных систем. В процессе работы, когда гостевая ОС обращается к памяти, её виртуальный адрес сначала преобразуется в guest-physical, а затем с помощью EPT – в физический адрес хоста.

Однако EPT требует наличия аппаратной поддержки, такой как Intel VT-x [26] или AMD-V [27,33] с поддержкой NPT, что ограничивает её доступность. Основные ограничения EPT включают дополнительные накладные расходы на управление двумя уровнями таблиц страниц и перехват доступа к памяти, что может замедлять работу системы, особенно при больших объёмах данных.

EPT используется для эффективного мониторинга нарушений прав доступа, таких как попытки чтения или записи в защищённые области памяти, но её использование ограничено на уровне целых страниц или диапазонов. Это делает EPT полезной для виртуализации, но менее эффективной для точного мониторинга мелких объектов памяти.

3.1.4 Сопоставление аппаратных механизмов

Пара watchpoints и permission faults покрывает два принципиально разных режима мониторинга доступа. Watchpoints дают адресную точность и позволяют получить событие на конкретный объект или узкий диапазон, однако ограничены малым числом аппаратных слотов и быстро становятся неэффективными при попытке масштабировать наблюдение. Permission faults, напротив, позволяют накрывать широкие области памяти за счёт страничной гранулярности, но событие возникает только при нарушении прав, поэтому механизм естественно подходит для режимов “first-touch” и редких триггеров, а при попытке превратить его в источник событий для каждого доступа накладные расходы растут пропорционально частоте обращений к выбранным страницам.

Таблица 1 — Сопоставление аппаратных механизмов мониторинга доступа к памяти и типичных областей применимости

Аппаратное средство	Охват адресного пространства	Профиль накладных расходов	Типичная область применимости
Watchpoints	Точечный	Низкие при редких срабатываниях; высокие при “горячих” адресах из-за частых исключений	Точная фиксация доступа к критичным объектам
PF (R/W/NX/XN/ PXN)	Широкий по страницам	Приемлемые при редких срабатываениях	Обнаружение первого срабатывания для регионов памяти
EPT	Виртуальное и физическое адресное пространство	Низкие, благодаря аппаратной поддержке	Мониторинг памяти в виртуал. системах, изоляция процессов

Таблица выше суммирует различия между аппаратными механизмами не в терминах их внутренней реализации, а с точки зрения практического выбора инструмента под задачу наблюдения, показывая, что решающим фактором становится не столько точность или охват сами по себе, сколько допустимый профиль накладных расходов и характер ожидаемых событий доступа.

3.2 Инструменты и механизмы операционных систем

Операционная система Linux предоставляет несколько классов механизмов наблюдения, которые позволяют строить мониторинг памяти без внедрения кода в целевой процесс. Содержательно такие механизмы удобно разделять на событийные источники контекста исполнения, средства наблюдения за состоянием страниц виртуальной памяти и подсистемы,

оценивающие активность диапазонов как “горячие” или “холодные”. В рамках инструмента мониторинга доступов к памяти эти механизмы используются не как взаимозаменяемые решения, а как комплементарные источники данных, которые совместно дают контекст, атрибуцию и управляемую точность наблюдения.

3.2.1 Событийная трассировка ядра и user-space: tracepoints/kprobes/ uprobes + eBPF

Событийная трассировка в Linux строится вокруг идеи, что многие существенные переходы состояния системы могут быть представлены в виде событий, на которые можно подписаться и получать структурированную информацию о происходящем. Tracepoints (точки трассировки ядра) являются статически заданными событиями, встроенными в код ядра, и по этой причине рассматриваются как наиболее стабильная база для наблюдения, поскольку их семантика фиксируется разработчиками ядра и сопровождается документацией [34]. В контексте мониторинга памяти такие события полезны для построения причинно-временной картины вокруг действий процесса, включая операции управления отображениями памяти и события, связанные с системными вызовами, которые задают границы фаз выполнения.

Kprobes (динамические пробники ядра) дополняют tracepoints как механизм, позволяющий подключаться к выбранным функциям ядра в тех случаях, когда статической точки трассировки нет или её данных недостаточно [35]. Такой подход расширяет наблюдаемость, однако повышает требования к инженерной аккуратности, потому что точки привязаны к конкретным адресам и символам и могут быть чувствительны к изменениям версий ядра. В практическом проектировании инструмента kprobes целесообразно рассматривать как средство адресного расширения покрытия, а tracepoints – как основу, на которую опирается штатный режим мониторинга.

Uprobes (динамические пробники пользовательского пространства) позволяют ставить события на инструкции или смещения в исполняемых файлах и разделяемых библиотеках целевого процесса, тем самым вводя семантические маркеры фаз выполнения без модификации исходного кода [36]. Это полезно, когда требуется связать признаки активности памяти не только с PID или TID, но и с логической фазой, например входом и выходом из аллокатора, обработчика запроса, криптографической процедуры или

сериализации данных. В результате uprobes формируют “шкалу контекста”, на которую затем накладываются наблюдения о состоянии страниц и активности диапазонов.

eBPF (extended Berkeley Packet Filter) реализует модель “выполнить фильтрацию в ядре”: пользовательское приложение загружает небольшую программу, которая после проверки верификатором исполняется в привязанной точке (tracepoint/kprobe/uprobe) и передаёт результаты в user-space. Обмен данными между eBPF-программами и пользовательским пространством выполняется через BPF maps (структуры хранения ключ–значение и специализированные типы), которые позволяют как сохранять агрегаты внутри ядра, так и выдавать результаты наружу без постоянной печати и форматирования в tracefs [37].

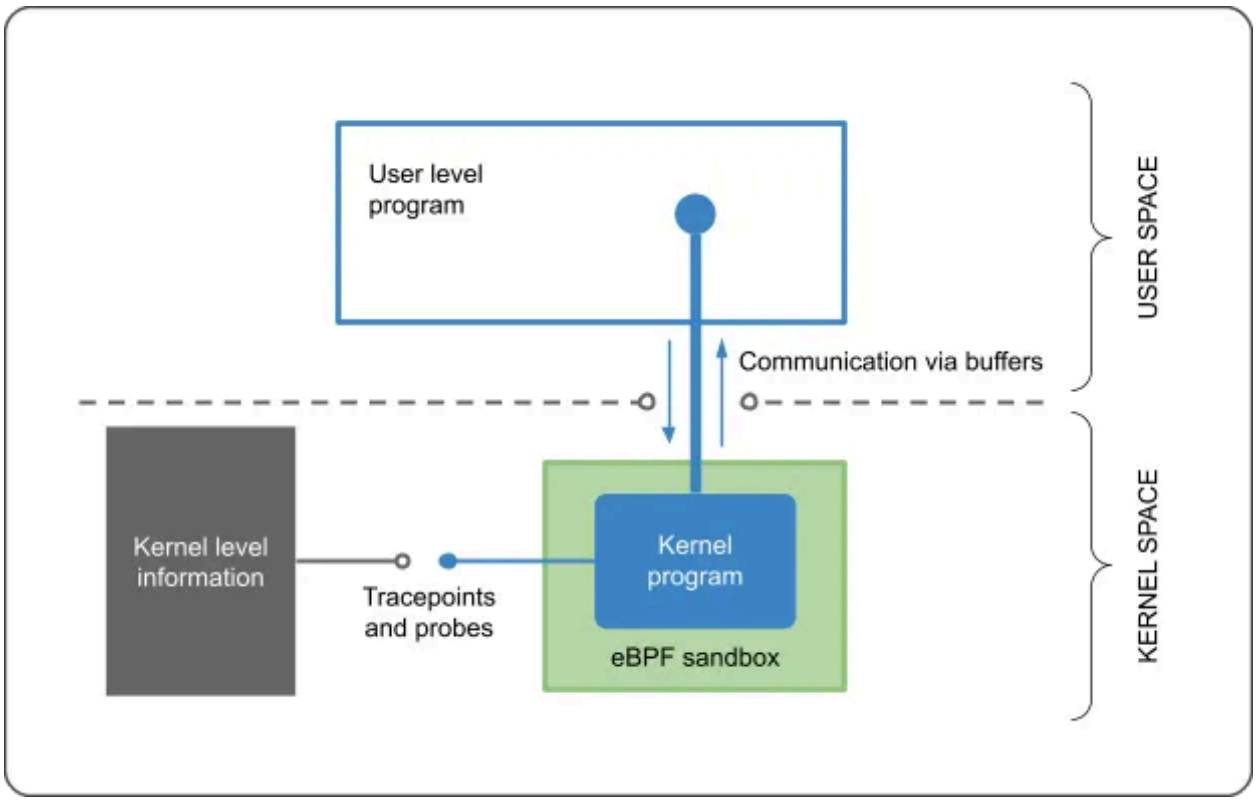

Рисунок 4 — Обобщённая схема работы eBPF при трассировке

За счёт этого возможна контекстная фильтрация по PID (Process Identifier), TID (Thread Identifier), cgroup (control group), а также привязка событий к syscall-контексту (системный вызов) и, при необходимости, сбор диагностических признаков вроде стека выполнения [38]. С инженерной точки зрения eBPF важен тем, что позволяет ограничивать объём данных, снижать накладные расходы за счёт агрегации “на месте” и реализовывать единый механизм включения и выключения интересующих источников событий в зависимости от режима наблюдения.

При использовании описанной связки следует учитывать, что событийная трассировка не является механизмом регистрации каждого обращения load/store к памяти. Её роль заключается в построении контекста исполнения и в фиксации событий, которые структурируют работу процесса и задают точки корреляции с данными о страницах и диапазонах, получаемыми другими механизмами.

3.2.2 Отслеживание модификаций страниц и атрибуция регионов: soft-dirty + pagemap

Soft-dirty в Linux – это механизм программного отслеживания записей в страницы виртуальной памяти на уровне записей таблиц страниц. Под страницей виртуальной памяти понимается фиксированный по размеру блок адресного пространства процесса (типично 4 KiB), а запись таблицы страниц (PTE, Page Table Entry) – это структура, в которой для конкретной виртуальной страницы хранится информация о её отображении и атрибутах доступа. Soft-dirty реализуется как специальный признак в PTE, который позволяет ответить на вопрос “выполнялась ли запись в эту страницу после момента последнего сброса признаков” [39].

Сброс soft-dirty выполняется через интерфейс /proc/<pid>/clear_refs, запись значения 4 означает “очистить soft-dirty” для страниц процесса [39–41]. Внутренне сброс не ограничивается обнулением бита. Документация ядра прямо указывает, что при очистке soft-dirty ядро также снимает признак writable у соответствующих PTE, временно делая страницы логически доступными только на чтение [39]. После этого первая попытка записи в любую из таких страниц порождает page fault (страничное исключение), а обработчик исключения в ядре восстанавливает writable и одновременно выставляет soft-dirty в PTE [39]. Ключевой смысл этого шага в том, что фиксируется именно факт первой записи в страницу в данном интервале, а не каждая отдельная операция store.

Такое поведение удобно трактовать как “срабатывание один раз на страницу в окне наблюдения”. Soft-dirty не хранит счётчик записей и не даёт идентификацию инструкции, которая выполнила запись, поэтому семантика остаётся бинарной: страница либо была модифицирована после сброса, либо нет. Важная инженерная деталь состоит в том, что возникающие после сброса page fault не означают подкачку или чтение с диска. В описанной

схеме страницы уже отображены в память, поэтому fault используется как управляемый механизм смены прав и установки диагностического признака, что и объясняет относительную “лёгкость” такого режима при умеренном числе модифицируемых страниц [39,42].

Интерфейс /proc/<pid>/pagemap предоставляет пользователю по одному 64-битному значению на каждую виртуальную страницу, описывая её состояние и ряд признаков. В частности, документация ядра и man-страница фиксируют, что бит 55 в pagemap соответствует soft-dirty и отражает состояние PTE для данной страницы [43,44]. Эти же источники полезны тем, что задают границы применимости: pagemap доступен только при включённой конфигурации CONFIG_PROC_PAGE_MONITOR и подчиняется правилам доступа, связанным с ptrace-режимом, а получение физического номера страницы (PFN, Page Frame Number) дополнительно ограничено привилегиями (CAP_SYS_ADMIN) [43,44]. ражения (mmap файлов и библиотек), анонимные диапазоны, стек и области, относящиеся к heap [45,46].

Отдельно важно учитывать динамику адресного пространства. Soft-dirty привязан к PTE и, следовательно, может теряться при операциях, которые разрушают и создают записи таблиц страниц заново (например, unmap/remap), а чтобы не допускать “тихой потери” информации о обновлённых регионах, в практических описаниях механизма подчёркивается, что при переотображении ядро может заново маркировать соответствующие области как soft-dirty [47]. Это не меняет базовой семантики “страница была записана после сброса”, но требует аккуратной интерпретации при активных перестроениях карт отображений процесса.

3.2.3 Непрерывный мониторинг “горячих” диапазонов: DAMON

DAMON (Data Access MONitor) – подсистема ядра Linux, предназначенная для оценки паттернов доступа к памяти на уровне диапазонов, включая частоту и интенсивность обращений, и для адаптивного уточнения границ наблюдаемых регионов [48]. В отличие от механизмов, ориентированных на события нарушения прав или на точечные триггеры, DAMON описывает поведение памяти как распределение активности по диапазонам, что делает его полезным для задач ранжирования и выбора областей, требующих повышенного внимания.

Результаты DAMON удобно рассматривать как вход для управляемой эскалации детализации наблюдения. При выявлении подозрительного региона или нетипичного изменения профиля активности инструмент может включать более детальную трассировку через eBPF и uprobes, сокращать окно наблюдения для повышения временного разрешения, а также переходить к точечным аппаратным триггерам через интерфейсы операционной системы, например perf_event_open (hardware breakpoints). Такой подход сохраняет практическую применимость мониторинга за счёт того, что высокозатратные режимы включаются ограниченно и адресно, а основная часть наблюдения выполняется средствами, рассчитанными на длительную работу и на обработку данных в агрегированном виде.

3.2.4 Сопоставление механизмов ОС

Таблица 2 — Сопоставление механизмов ОС для мониторинга доступа к памяти и профиля накладных расходов

Механизм ОС	Охват адресного пространства	Профиль накладных расходов	Типичные области применимости
Tracepoints/ kprobes/uprobes + eBPF	Событие / точка в коде	Низкий–умеренный; основная стоимость определяется частотой событий	Построение контекста вокруг памяти: mmap/ munmap/mprotect, ошибки, I/O-фазы, фазовые маркеры
Soft-dirty	Широкий по выбранным VMA; страничная гранулярность (типично 4 KiB)	Низкий при умеренном числе записываемых страниц;	Длительное наблюдение “куда писали” и выделение регионов для углубления анализа
DAMON	Широкий по диапазонам (региональная модель; границы уточняются ядром)	Низкий–умеренный; рост при увеличении диапозона	Выбор “горячих” областей памяти, где имеет смысл повышать детализацию

На практике наиболее стабильная схема для attach-модели выглядит как “контекст через trace/eBPF + следы записей через soft-dirty/pagemap + выбор горячих регионов через DAMON”, а точечные триггеры через perf_event_open используются как режим эскалации на коротком интервале, когда уже известен конкретный адрес или узкий набор страниц. Такое разделение ролей позволяет удерживать накладные расходы на приемлемом уровне и при этом иметь понятный путь повышения точности, когда наблюдение выявило конкретный кандидат для углублённой проверки.

3.3 Инструменты на уровне приложений

Инструменты мониторинга памяти на уровне приложений ориентированы на наблюдение поведения конкретной программы в пользовательском пространстве и, в зависимости от класса, либо фиксируют нарушения корректности обращений к памяти, либо собирают профиль распределения памяти, либо анализируют трассы обращений на уровне отдельных инструкций.

Ключевое различие между такими средствами заключается в том, что именно считается наблюдаемым событием и какая гранулярность доступна пользователю: нарушение доступа, вызов аллокатора, либо обращение к памяти на уровне load-инструкции.

Для сопоставления ниже рассмотрены несколько показательных представителей, которые покрывают эти классы и хорошо иллюстрируют компромиссы между полнотой наблюдения, интерпретируемостью результатов и накладными расходами.

3.3.1 AddressSanitizer как инструмент детектирования нарушений memory-safety

AddressSanitizer (ASan, Address Sanitizer) относится к классу санитайзеров, реализующих компиляторную инструментализацию для выявления нарушений корректности обращений к памяти. Механизм работы в базовой форме опирается на теневую память (shadow memory) и на внедрённые проверки, которые исполняются вместе с программой и сопоставляют фактический адрес обращения с допустимым диапазоном целевого объекта. Для повышения надёжности детектирования вокруг объектов формируются защитные зоны (redzones), а освобождённые области на время удерживаются в карантине, что позволяет эффективнее выявлять use-after-free, когда обращение происходит вскоре после освобождения [49].

Событием для ASan является обнаружение обращения, противоречащего модели допустимых адресов, то есть ошибка фиксируется в момент выполнения проблемной операции. Результат наблюдения представлен диагностическим отчётом, в который обычно входят адрес нарушения, тип дефекта и контекст исполнения, причём контекст чаще всего задаётся стеком вызовов (stack trace), позволяющим привязать ошибку к месту в исходном коде или к символам бинаря. [49]

Типичные сценарии использования ASan связаны с тестированием и регрессионной проверкой, где важнее систематическая обнаруживаемость дефектов, чем сохранение исходного профиля производительности. Существенным ограничением является необходимость собирать целевой бинарный файл с включённой инструментализацией и учитывать изменение профиля потребления ресурсов из-за проверок и поддержания метаданных. [49]

3.3.2 Valgrind Memcheck как эталонный dynamic checking на базе DBI

Valgrind представляет собой фреймворк динамической бинарной инструментализации (DBI, Dynamic Binary Instrumentation), в рамках которого исполнение программы переводится в управляемую форму и сопровождается добавлением проверок и учётом дополнительных состояний. [50] Memcheck является наиболее известным инструментом поверх Valgrind, предназначенным для выявления широкого класса ошибок работы с памятью, включая обращения к невалидным адресам и использование неинициализированных данных. [51]

Функциональная особенность Memcheck состоит в том, что он поддерживает теневое представление состояния памяти, отслеживая адресуемость и инициализацию на уровне выполнения, что делает его диагностику особенно полезной в сценариях, где дефект проявляется не сразу и зависит от конкретной траектории исполнения. [51] События в Memcheck возникают при выполнении операции, нарушающей отслеживаемые инварианты, а отчёт обычно включает описание типа дефекта и привязку к месту выполнения с контекстом, достаточным для практической локализации причины.

Компромисс DBI-подхода заключается в высоких накладных расходах, поскольку существенная часть исполнения проходит через слой инструментализации и сопровождается обновлением теневого состояния. [50] По этой причине Memcheck принято рассматривать как эталон точности для подтверждения гипотез о дефектах, а не как инструмент длительного мониторинга в условиях чувствительности к производительности.

3.3.3 Heaptrack как профилирование динамических выделений памяти

heaptrack относится к инструментам профилирования динамических выделений, то есть к средствам, которые наблюдают события распределения памяти аллокатором и агрегируют информацию о выделениях и освобождениях [52]. В отличие от средств класса memory-safety checking, здесь наблюдаемой единицей является не обращение к произвольному адресу, а событие выделения или освобождения, которое имеет понятную семантику на уровне приложения и естественно связывается с точками вызова (call sites).

Основной практический эффект heaptrack заключается в высокой интерпретируемости результатов. Так как наблюдаются события аллокаций, инструмент позволяет выявлять источники роста памяти, пики и удержание, связывая их с конкретными цепочками вызовов и компонентами программы [52]. Подобный профиль особенно полезен, когда требуется объяснить изменение потребления памяти через логику работы приложения, а не через низкоуровневые адреса страниц.

Ограничение данного класса инструментов состоит в том, что они не предназначены для регистрации обращений load/store к уже выделенным данным и, следовательно, не дают прямого ответа на вопросы о паттернах чтения и записи внутри объектов, если эти паттерны не проявляются через дополнительные аллокации или освобождения.

3.3.4 MemGaze как анализ трасс обращений к памяти на уровне load-инструкций

MemGaze относится к исследовательским инструментам анализа трасс обращений к памяти на уровне load-инструкций, то есть операций чтения данных из памяти, представимых в виде последовательности событий “инструкция прочитала данные по адресу A” с привязкой к контексту выполнения. Ключевая идея MemGaze состоит в том, чтобы получать детализированный адресный след чтений без перехода к полностью программной эмуляции или тотальной динамической инструментализации, которые обычно резко увеличивают накладные расходы. [53]

Метод MemGaze опирается на аппаратную трассировку исполнения Intel PT (Intel Processor Trace) и на вставку в поток трассировки специальных программных маркеров через инструкцию ptwrite. Intel PT

обеспечивает высокоскоростной сбор информации о ходе выполнения (прежде всего о переходах управления), а ptwrite позволяет добавлять в этот поток дополнительные значения, порождённые программой, в частности адреса, относящиеся к интересующим обращениям к памяти [53]. За счёт такого совмещения аппаратного канала трассировки и программных маркеров MemGaze реализует “облегчённую” трассу обращений: фиксируется не каждый микрошаг работы процессора, а выбранный слой событий, достаточный для анализа поведения чтений данных.

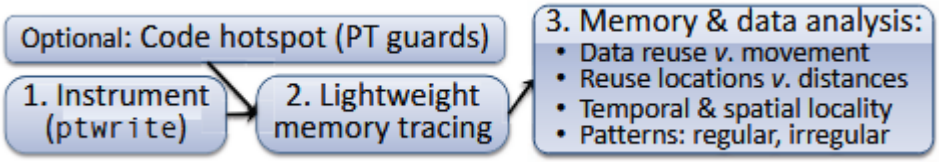

Рисунок 5 — Схема работы MemGaze

Типовой конвейер MemGaze можно описать как последовательность из трёх стадий [53]. На первой стадии выполняется инструментирование, где в выбранные точки исполнения добавляются операции, передающие необходимые значения (например, адреса чтения) в поток трассировки через ptwrite. [53] На второй стадии собирается lightweight memory trace, то есть компактный поток событий, который извлекается из PT-данных и затем декодируется в последовательность обращений, пригодную для аналитической обработки. [53] На третьей стадии выполняется memory and data analysis, где по трассе вычисляются характеристики повторного использования и перемещения данных, свойства временной и пространственной локальности, а также выявляются регулярные и нерегулярные паттерны обращений. [53]

Событие в данной постановке соответствует обращению чтения, а ценность подхода определяется тем, что адресные трассы позволяют анализировать поведение памяти в терминах паттернов и повторного использования, которые принципиально трудно восстановить только из профилей аллокаций или из редких диагностических срабатываний. [53] Ограничения подобных решений связаны с требованиями к среде трассировки и объёму обрабатываемых данных, поскольку переход к гранулярности уровня инструкций генерирует существенно более насыщенный поток наблюдений, чем аллокационное профилирование или отчёты о дефектах memory-safety.

3.3.5 Сравнение инструментов

Рассмотренные инструменты естественно формируют три разных уровня наблюдаемости:

1) ASan и Memcheck ориентированы на детектирование нарушений корректности обращений к памяти и дают наиболее прямую диагностику дефектов memory-safety, но различаются способом достижения эффекта: ASan делает это через компиляторную инструментализацию, а Memcheck – через выполнение под DBI-слоем [49–51];
2) heaptrack фиксирует события аллокации и освобождения и потому лучше отвечает на вопросы об источниках роста потребления памяти и удержания объектов, чем на вопросы о реальных паттернах чтения и записи внутри выделенных областей [52];
3) MemGaze ориентирован на анализ адресных трасс на уровне load-событий, что делает его наиболее информативным для исследований паттернов доступа, но одновременно повышает требования к условиям трассировки и к обработке значительных объёмов данных [53].

Таблица 3 — Сопоставление гранулярности наблюдения, доступного контекста и профиля накладных расходов

Название инструмента	Тип наблюдаемых событий	Доступный контекст	Профиль накладных расходов
ASan	Ошибки memory-safety	Стек вызовов, адрес нарушения, тип дефекта	Умеренный; зависит от плотности проверок и объёма данных
Valgrind Memcheck	Ошибки памяти при выполнении	Детализированная диагностика с привязкой к месту выполнения, типу дефекта и адресу	Высокий; определяется DBI и поддержанием теневого состояния
heaptrack	Call site аллокации/ освобождения	Цепочка вызовов и агрегаты по точкам выделения	Низкий–умеренный; определяется частотой аллокаций и глубиной сбора стека
MemGaze	Обращение load-уровня (load-level trace)	Адресный след чтений и производные метрики локальности/ reuse	Переменный; зависит от режима трассировки Intel PT/ptwrite

Таблица 4 — Сопоставление модели внедрения, применимости без модификации процесса и сравнение типичных сценариев

Название инструмента	Модель внедрения	Без изменения целевого процесса	Типичные сценарии использования
ASan	Пересборка с инструмента-лизацией	Нет	CI и регрессионные тесты, fuzzing
Valgrind Memcheck	Запуск под рантаймом DBI	Частично (запуск под инструментом)	Точная отладка
heaptrack	Запуск в подготовленном окружении	Частично (требуется контроль окружения запуска)	Поиск причин роста памяти, пики, удержание
MemGaze	Инструменти-рование + сбор PT-трассы и последующий анализ	Частично (зависит от условий трассировки и режима запуска)	Анализ паттернов чтения, локальности и повторного использования

Сопоставление в таблицах показывает характерную зависимость между точностью анализа и скоростью работы инструментов мониторинга памяти на уровне приложений.

При повышении точности обычно растёт частота и “близость” вмешательства в путь исполнения, а вместе с этим увеличиваются накладные расходы. Memcheck, который строит модель адресуемости и инициализации на уровне динамической бинарной инструментализации, даёт наиболее детальную и универсальную диагностику, но сопровождается высокой стоимостью из-за постоянного инструментирования инструкций и обновления теневого состояния. [50,51]

В практическом обзоре эти инструменты целесообразно воспринимать как взаимодополняющие подходы. Они закрывают разные классы задач и демонстрируют разные компромиссы между точностью диагностики, объяснимостью результатов и накладными расходами, что задаёт основу для

выбора критериев и требований к системам мониторинга памяти в дальнейших разделах.

3.4 Выводы

В этой главе был проведён сравнительный анализ различных инструментов и технологий для мониторинга памяти, охватывающих как аппаратные механизмы, так и программные решения, встроенные в операционные системы и приложения. Инструменты мониторинга памяти можно классифицировать на:

- аппаратные механизмы (watchpoints, permission faults) – для точечной диагностики и мониторинга доступа к памяти с высокой точностью, но с ограничениями по числу слотов и накладным расходам при масштабировании. Extended Page Tables (EPT) контролируют доступ к памяти на уровне гипервизора, что позволяет эффективно управлять виртуальной памятью в виртуализированных системах;
- механизмы ОС (tracepoints, kprobes, eBPF, soft-dirty, DAMON) – для мониторинга состояния памяти и активности процессов с контекстной фильтрацией, позволяя выявить «горячие» области и адаптивно увеличивать детализацию;
- инструменты на уровне приложений (ASan, Memcheck, heaptrack, MemGaze) – для детектирования нарушений работы с памятью и анализа паттернов доступа с учетом производительности и накладных расходов.

Внутри каждой из этих категорий инструменты были сравнены между собой по различным параметрам, таким как точность, накладные расходы и область применения. Результаты сравнений представлены в таблицах, включая таблицу сравнения аппаратных механизмов (таблица 1), таблицу сравнения механизмов ОС (таблица 2) и таблицу сравнения инструментов на уровне приложений (таблица 3), что позволяет сделать обоснованный выбор в зависимости от целей мониторинга.

В результате проведенного анализа инструментов и технологий мониторинга памяти можно выделить несколько ключевых выводов:

Во первых, чем более детализированное наблюдение требуется (например, на уровне инструкций), тем выше накладные расходы. Memcheck и Valgrind предлагают высокую точность, но имеют значительные затраты

ресурсов. Heaptrack и MemGaze предлагают более легкие альтернативы для профилирования памяти, но с меньшей точностью.

Во вторых, для эффективного мониторинга лучше всего использовать комбинацию различных инструментов, в зависимости от конкретных целей и задач. Например, tracepoints и eBPF обеспечивают контекстную фильтрацию, а watchpoints и permission faults могут быть использованы для точечных триггеров.

4 Формирование требований к инструменту

В предыдущих главах были определены ключевые потребности и типовые задачи, решаемые с помощью инструмента мониторинга памяти. На основе анализа задач мониторинга доступа к памяти и обзора существующих решений (проведённых в предыдущих главах) выделены три типа требований: пользовательские, функциональные и эксплуатационные. Каждый тип охватывает свой аспект. Пользовательские требования отражают потребности и ожидания конечных пользователей относительно возможностей инструмента. Функциональные требования детализационно описывают, какие конкретные функции и характеристики должен реализовать инструмент, чтобы удовлетворить пользовательские потребности. Эксплуатационные требования устанавливают внешние условия применения и ограничения, в рамках которых инструмент должен функционировать. Такая структура позволяет перейти от высокоуровневых целей к конкретным спецификациям и учесть контекст использования инструмента.

4.1 Пользовательские требования

Пользовательские требования определяют, какие задачи должен решать инструмент мониторинга памяти с точки зрения пользователя. Эти требования выведены из анализа практических задач мониторинга и ограничения современных средств. Ниже перечислены основные потребности:

Мониторинг операций доступа к памяти. Инструмент должен отслеживать фундаментальные операции доступа в адресном пространстве процесса – чтение, запись и исполнение. Это позволит пользователю фиксировать все важные события, связанные с памятью целевого процесса, в реальном времени.

Контекстная фильтрация событий. Инструмент должен обеспечивать возможность фильтрации наблюдаемых событий по контексту выполнения. В частности, пользователь должен иметь возможность отбирать события по конкретному потоку выполнения, загруженному модулю или библиотеке, инициирующему системному вызову и диапазону виртуальных адресов памяти. Такая фильтрация поможет сосредоточиться на наиболее релевантных данных и уменьшить объем лишней информации.

Атрибуция событий к контексту. Инструмент должен предоставлять для каждого зафиксированного события доступа к памяти информацию о его контексте: например, идентификатор потока и модуля, в котором произошло событие, адрес или диапазон адресов, тип операции (read/write/execute) и при необходимости связанный системный вызов. Наличие контекстной информации для каждого события повышает интерпретируемость результатов и позволяет пользователю связывать обнаруженные аномалии с конкретными частями программы.

Отсутствие необходимости модификации приложения. Инструмент должен работать без требовательности изменения исходного кода или бинарного файла отслеживаемого приложения. Пользовательское программное обеспечение, для которого проводится мониторинг, не должно требовать встроенной поддержки или специальной компиляции – инструмент должен подключаться к уже скомпилированному приложению внешними средствами (например, через механизмы операционной системы), не нарушая его работу.

Конфигурируемость и удобство использования. Инструмент должен предоставлять понятный способ указать целевой процесс для мониторинга и задать необходимые параметры наблюдения. Пользователь должен иметь возможность легко запустить или остановить мониторинг, а также настроить критерии фильтрации (например, через командную строку или файл конфигурации) без сложной ручной настройки в исходном коде инструмента.

Понятный вывод данных. Результаты мониторинга должны предоставляться в формате, удобном для анализа. Инструмент должен выводить собранные события в ясном виде – например, в виде журнала (лог-файла) или консольного отчета, где по каждой операции указаны ее параметры и контекст. Это позволит пользователю проанализировать последовательность событий, выявить аномальные паттерны доступа и соотнести их с выполняемой программой.

4.2 Функциональные требования

Функциональные требования фиксируют наблюдаемое поведение и возможности инструмента мониторинга памяти, необходимые для реализации сформулированных пользовательских запросов. Ниже приведён перечень основных функциональных требований к разрабатываемому инструменту:

Регистрация событий активности памяти. Инструмент должен регистрировать события выбранного класса активности памяти целевого процесса в режиме online. Под классом активности памяти понимается тип наблюдаемых событий, определяющий гранулярность и семантику регистрации (например, события модификации страниц, обращения к заданным диапазонам, изменения карты виртуальной памяти и т.п.). События должны регистрироваться с сохранением их порядка возникновения и с возможностью последующего вывода в потоковом режиме или сохранения в файл. Поддержка регистрации событий, связанных с попытками исполнения из выбранных регионов памяти, допускается как расширение.

Состав и структура записи о событии. Для каждого зарегистрированного события инструмент должен формировать запись фиксированного формата. Запись о событии должна включать: идентификатор процесса, виртуальный адрес события или границы диапазона (если событие агрегировано по диапазону), тип события (принадлежность к выбранному классу активности), временную метку или порядковый номер события, идентификатор потока, привязку адреса к области виртуальной памяти (модуль/сегмент/регион), если такая привязка определяется.

Фильтрация по идентификатору потока. Инструмент должен предоставлять возможность фильтрации событий по идентификатору потока исполнения целевого процесса. При включённой фильтрации инструмент должен регистрировать и/или выводить только те события, которые относятся к указанному набору идентификаторов потоков.

Фильтрация по модулю программы или области памяти. Инструмент должен предоставлять фильтрацию событий по принадлежности адреса события к заданному модулю (например, загруженной библиотеке) или к заданной области виртуальной памяти. Пользователь должен иметь возможность задавать критерий фильтрации как: имя/идентификатор модуля или диапазон виртуальных адресов или тип области виртуальной памяти (например, стек, куча), если такая классификация поддерживается выбранным способом определения регионов. Инструмент должен сопоставлять адрес события с описанием регионов виртуальной памяти и применять фильтр согласно заданным условиям.

Комбинация критериев фильтрации. Инструмент должен позволять одновременное применение нескольких фильтров. Событие регистрируется и/или выводится тогда и только тогда, когда удовлетворяет всем активным

фильтрам (логическое AND). В рамках одного фильтра допускается задание набора значений (логическое OR), например списка потоков, модулей или диапазонов адресов. Это требование направлено на обеспечение максимальной гибкости инструмента, чтобы он мог быть адаптирован под узко определённые случаи использования.

Режимы запуска. Инструмент должен поддерживать два режима работы: запуск целевого приложения под контролем инструмента (launch) и подключение к уже запущенному процессу (attach). В обоих режимах мониторинг должен выполняться без модификации и пересборки целевого приложения; инструмент рассматривается как внешнее средство наблюдения, применимое к бинарному файлу или запущенному процессу «как есть».

Вывод и хранение результатов мониторинга. Инструмент должен предоставлять результаты наблюдения в удобной форме для анализа. Практически это означает, что собранные события доступа к памяти (после применения всех фильтров) должны быть либо выводимы в консоль/файл в структурированном формате (например, в виде таблицы или журнала с колонками: время, поток, адрес, тип доступа, модуль и пр.), либо доступны через API для последующей обработки. Данный функционал гарантирует, что пользователь сможет проанализировать и использовать собранные данные. Важно предусмотреть, чтобы объём выводимой информации мог быть значительным, поэтому инструмент должен иметь опцию буферизации или сохранения результатов на диск для дальнейшего изучения.

Перечисленные функциональные требования в совокупности описывают поведение инструмента, достаточное для выполнения запросов пользователя. Их реализация позволит решить основную задачу – обеспечить контекстный мониторинг памяти с нужной гибкостью и точностью. Следующим шагом является рассмотрение эксплуатационных требований, накладывающих ограничения и условия на функционирование инструмента.

4.3 Эксплуатационные требования

Эксплуатационные требования отражают внешние условия работы инструмента и ограничения, которые необходимо учесть при его разработке и внедрении. Эти требования дополняют функциональные, гарантируя, что инструмент будет работоспособен в заданном окружении и соответствовать

практическим ограничениям. К основным эксплуатационным требованиям относятся:

Операционная среда. Инструмент должен функционировать в операционной системе Linux и использовать стандартные механизмы ОС для перехвата и отслеживания событий (например, возможности ptrace, eBPF, tracepoints и др.). Для работы инструмента могут требоваться повышенные привилегии (права администратора), так как мониторинг процессов на уровне ядра и памяти обычно ограничен для непривилегированных пользователей. Инструмент должен быть совместим как с архитектурами x86, так и с ARM.

Совместимость с целевыми приложениями. Инструмент должен работать с широким спектром пользовательских приложений, не предъявляя к ним специальных требований. В частности, не предполагается, что целевые программы будут скомпилированы с особыми флагами или запущены в особом режиме; инструмент совместим как с отладочными, так и с релизными сборками приложений. Ограничением (в рамках области исследования) может являться поддержка только пользовательского пространства: то есть мониторинг ядра или драйверов не рассматривается, инструмент работает лишь с адресным пространством обычных процессов.

Производительность и накладные расходы. Инструмент должен функционировать с приемлемыми накладными расходами по времени и памяти, чтобы его использование было возможно в реальных сценариях. Под «приемлемыми» понимается, что при активированном мониторинге замедление работы целевого приложения минимально возможное, не изменяющее существенно его поведение. Конкре тный целевой потолок накладных расходов зависит от выбранных технологий (например, использование чисто программных перехватчиков может давать замедление на порядок, тогда как аппаратная поддержка – на проценты), однако инструмент в любом случае не должен потреблять чрезмерно много ресурсов (ЦП, памяти) или генерировать неподъёмный объём данных. Данное требование соответствует ожиданию более низкой ресурсоёмкости по сравнению с существующими профилировщиками, достигающими аналогичного уровня детализации.

Надежность и безопасность работы. Инструмент должен быть устойчивым к сбоям и не нарушать стабильность целевого процесса. В ходе длительного мониторинга не должно происходить утечек ресурсов (например, памяти) или накопления ошибок, способных привести к падению

инструмента или приложения. Кроме того, работа инструмента не должна нарушать целостность системы: он не должен вносить изменения в адресное пространство процесса, кроме необходимых для наблюдения, и не должен создавать уязвимостей в безопасности ОС.

Перечисленные эксплуатационные требования обеспечивают соответствие инструмента условиям реального применения. Они гарантируют, что разработанное решение будет работать в заданной программно-аппаратной среде, не нарушая ее ограничений и удовлетворяя практическим критериям по производительности и надёжности.

4.4 Выводы

В данной главе на основе результатов предыдущего анализа были сформированы требования к инструменту мониторинга памяти, разделённые на три категории. Пользовательские требования, выведенные из изученных задач мониторинга и недостатков существующих решений, определяют, какие функции ожидает получить специалист при использовании инструмента. На их основании были получены функциональные требования, детализирующие необходимые механизмы и особенности реализации. Кроме того, сформулированы эксплуатационные требования, отражающие внешние условия работы и ограничения инструмента – среду выполнения, требования к ресурсам и стабильности.

Таким образом, сформированный комплект требований задаёт основу для проектирования архитектуры и последующей разработки инструмента. Пользовательские и функциональные требования обеспечивают ориентиры для реализации ключевых возможностей (контекстная фильтрация, перехват событий и др.), тогда как эксплуатационные требования подчёркивают важность учёта ограничений среды (ОС, аппаратных ресурсов) и требований к эффективности. Выполнение всех указанных требований в совокупности должно привести к созданию инструмента, удовлетворяющего цели исследования и превосходящего по ряду показателей существующие решения мониторинга памяти.

5 Проектирование архитектуры на концептуальном уровне

Проектирование архитектуры инструмента мониторинга памяти является ключевым этапом, переходящим от требований (сформулированных в главе 3) к конкретному решению, позволяющему эти требования реализовать. Целью архитектурного проектирования на концептуальном уровне является определение структуры системы – набора модулей и взаимодействий между ними – таким образом, чтобы обеспечить выполнение всех функциональных возможностей, ожидаемых от инструмента. Архитектура разрабатываемого средства будет модульной, что означает логическое разделение системы на уровни с чётко очерченными ответственностями. . Данный подход обеспечивает изоляцию низкоуровневых механизмов перехвата событий от высокоуровневого представления данных пользователю, упрощает модификацию и расширение системы. В контексте поставленных требований это означает, что каждому важному аспекту функциональности будет соответствовать отдельный модуль или подсистема.

5.1 Блок-схема алгоритма работы инструмента мониторинга памяти

Прежде всего, рассмотрим общий алгоритм работы инструмента мониторинга памяти на концептуальном уровне. Ниже описана последовательность шагов, которую выполняет система при запуске мониторинга, включая два варианта начала работы: запуск целевого приложения под контролем инструмента (режим launch) и подключение к уже запущенному процессу (режим attach). В обоих случаях пользователь инициирует сеанс мониторинга через интерфейс, указывая необходимый режим и параметры (идентификатор процесса или путь к исполняемому файлу, а также критерии фильтрации событий). Инструмент либо запускает процесс под своим контролем, либо подключается к существующему, после чего настраивает средства перехвата событий памяти.

Далее целевой процесс исполняется, и инструмент в режиме реального времени перехватывает события доступа к памяти (чтение, запись, исполнение инструкций). Каждое событие проходит через блок фильтрации: проверяется, соответствует ли оно заданным пользователем условиям (по потоку, модулю, адресу и т.п.). Если событие удовлетворяет активным фильтрам, инструмент осуществляет регистрацию (запись) события – сохраняет информацию о нем в

структуру результатов и/или выводит пользователю согласно настройкам. Если событие не попадает под заданные критерии, оно отбрасывается и не нагружает вывод.

Такой цикл перехвата и фильтрации повторяется для всех поступающих событий, пока пользователь не остановит мониторинг. При получении команды остановки инструмент отключается от процесса (или завершает запущенный процесс, если работал в режиме launch) и формирует отчёт.

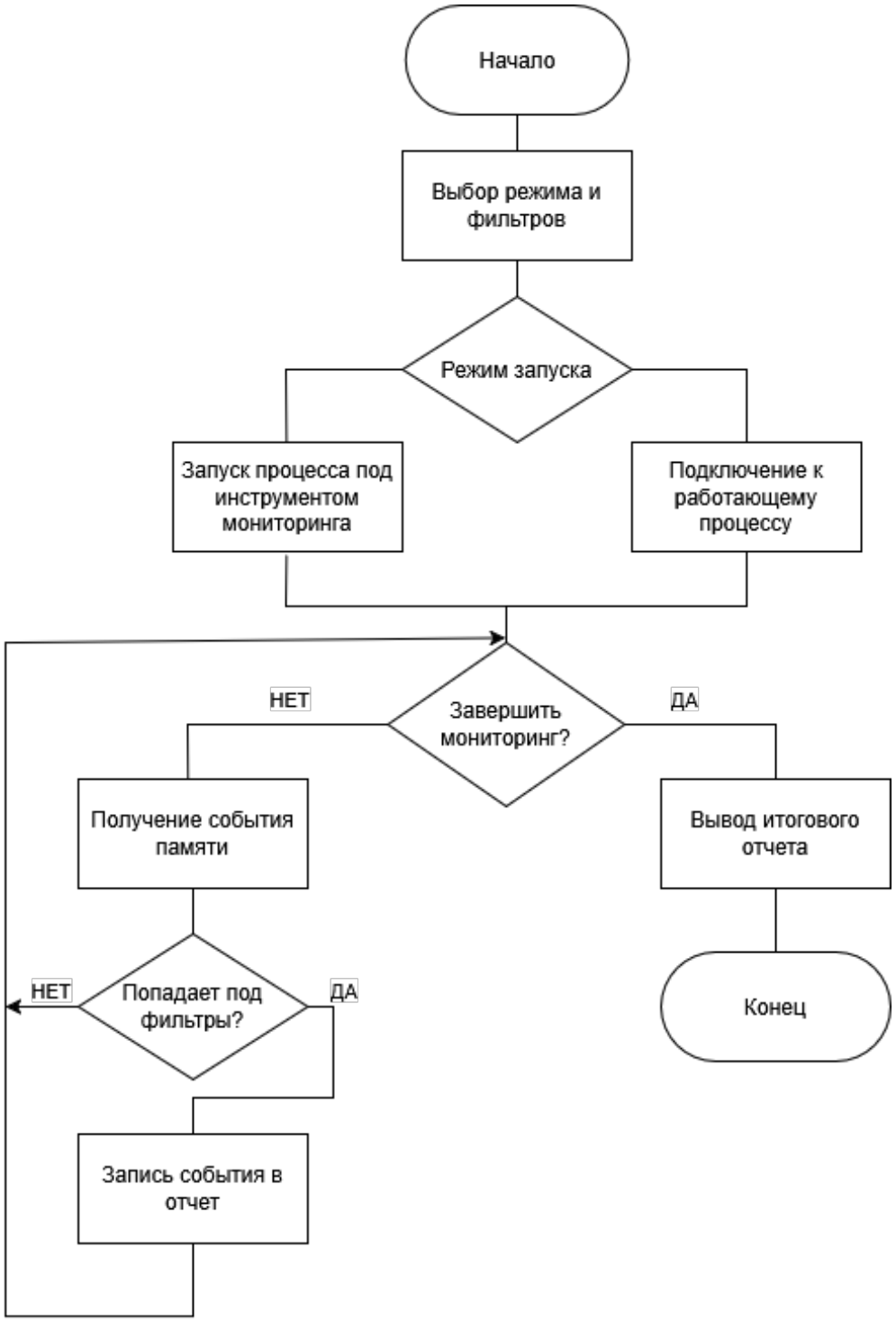

Рисунок 6 — Блок-схема алгоритма работы инструмента мониторинга памяти

5.2 Основные модули планируемой архитектуры

Исходя из функциональных требований, сформулированных ранее, разработанная архитектура разделяется на несколько основных модулей, каждый из которых отвечает за свой участок работы инструмента. Такая организация позволяет отнести эти компоненты к различным логическим уровням системы – от низкоуровневого взаимодействия с ОС до интерфейса с пользователем. Ниже перечислены основные модули предлагаемой архитектуры, их функции и положение в общей иерархии:

Модуль перехвата событий. Отвечает за перехват событий доступа к памяти целевого процесса. Этот модуль инициализирует мониторинг: в режиме launch запускает приложение под отладочным контролем, в режиме attach – подключается к уже работающему процессу. Далее модуль перехвата устанавливает необходимые механизмы слежения за памятью, чтобы получать уведомления о событиях. Перехваченные события в сыром виде (с указанием адреса, типа доступа, идентификаторов процесса и потока) передаются на следующий уровень обработки. Модуль реализует функциональное требование **регистрации событий активности памяти**, обеспечивая фиксацию всех происходящих событий заданного класса.

Модуль фильтрации. Реализует контекстную фильтрацию событий согласно критериям, заданным пользователем. Он принимает события (вместе с их атрибутами) и проверяет, удовлетворяют ли они активным фильтрам: по идентификатору потока, по принадлежности адреса к определённому модулю или диапазону памяти, по типу события и другим условиям (как указано в требованиях, фильтрация должна допускать комбинацию критериев с логическим AND/OR). Модуль фильтрации отбрасывает события, не соответствующие условиям, предотвращая их попадание в результирующий вывод, и пропускает дальше только релевантные данные. Таким образом, реализуются функциональные требования по фильтрации по потоку, модулю, адресу, а также требование поддержки комбинации фильтров (одновременное применение нескольких критериев).

Модуль контекстной атрибуции – также относится к уровню обработки событий и тесно взаимодействует с модулем фильтрации (возможно, интегрируясь с ним в рамках общего процесса обработки). Его функция – обогащение каждого перехваченного события дополнительной контекстной информацией, повышающей ценность данных мониторинга. В частности,

модуль атрибуции определяет, к какому компоненту или области памяти относится адрес события: например, сопоставляет адрес с загруженным модулем или сегментом (код, куча, стек и т.д.), идентифицирует библиотеку или исполняемый файл, откуда произошёл доступ. Эта информация затем используется модулем фильтрации (для принятия решения) и модулем хранения результатов (для формирования отчёта). Модуль контекстной атрибуции реализует потребность в привязке события к контексту выполнения, подчеркнутую в пользовательских требованиях, и формирует полный состав записи о событии

Модуль хранения результатов – отвечает за накопление и структурирование данных о событиях, прошедших через фильтр, и относится к уровню управления данными системы. Он сохраняет отфильтрованные события в требуемом формате и предоставляет их для вывода. В простейшем случае данный модуль может формировать журнал (лог-файл) или таблицу событий, включающую ключевые поля (время, адрес, тип доступа, контекст и пр.), либо хранить данные в памяти до запроса.

Модуль пользовательского интерфейса – расположен на верхнем уровне архитектуры (уровень взаимодействия с пользователем) и обеспечивает интерфейс управления и представления данных. Он предоставляет средства для конфигурирования инструмента и отображения результатов. С одной стороны, интерфейс позволяет пользователю задать все необходимые параметры мониторинга: указать целевой процесс (имя исполняемого файла для запуска или PID для подключения), выбрать режим работы (launch или attach), а также установить критерии фильтрации (например, через опции командной строки, файл конфигурации или интерактивные команды). Это удовлетворяет пользовательские требования по конфигурируемости и удобству использования инструмента без необходимости менять исходный код целевого приложения. С другой стороны, модуль интерфейса отвечает за вывод собранной информации пользователю: он получает структурированные данные от модуля хранения и отображает их в понятном виде – например, выводит на консоль таблицу событий или генерирует отчет.

Логическая организация модулей по уровням такова, что поток данных идет снизу вверх: от перехвата низкоуровневых событий к их фильтрации и обогащению контекстом, затем к сохранению и отображению. Одновременно команды и настройки идут от пользователя сверху вниз: интерфейс инициирует запуск/подключение и задаёт параметры для глубинных модулей. Таким

образом, модульно-слойная архитектура обеспечивает ясное разграничение: каждый уровень решает свой класс задач и взаимодействует с соседними слоями через чётко определённые интерфейсы.

5.3 Взаимодействие модулей

На основе архитектуры, описанной выше, можно проследить, как данные и управляющие команды проходят через систему от начала до завершения мониторинга. Ниже представлена диаграмма последовательности, демонстрирующая обмен сообщениями между пользователем и основными модулями инструмента во времени.

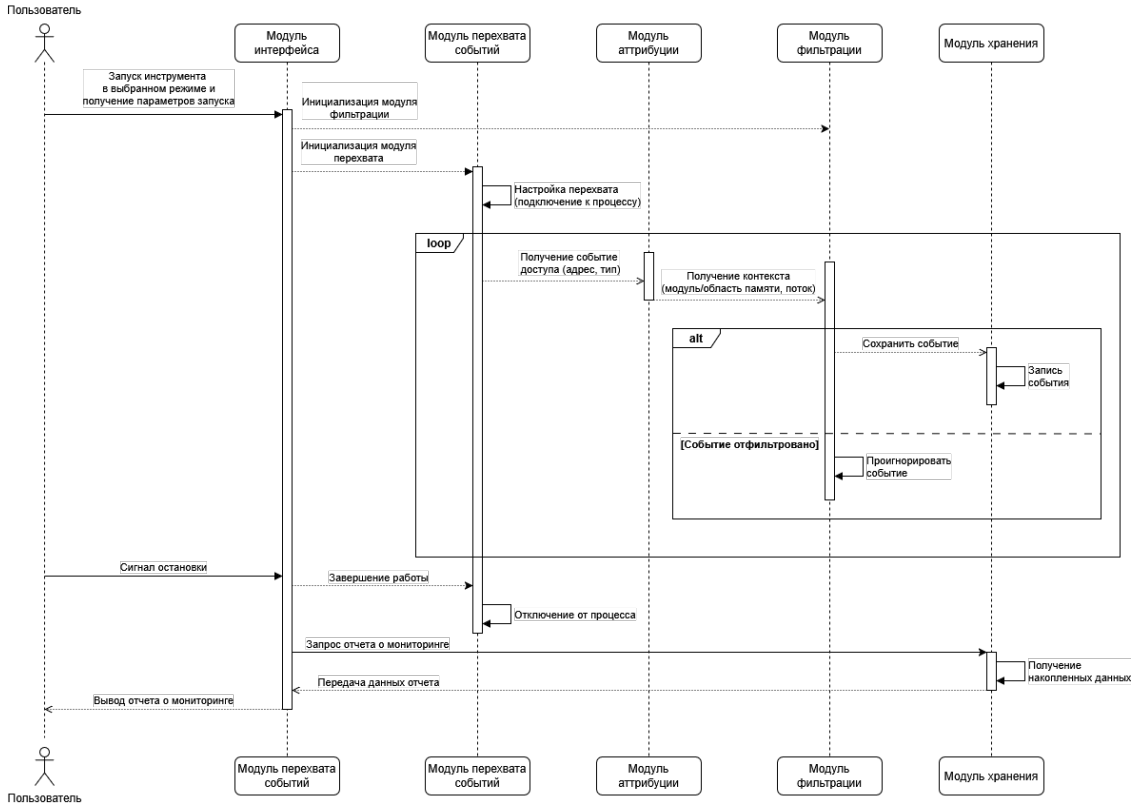

Рисунок 7 — UML-диаграмма последовательности взаимодействия

Диаграмма последовательности отображает детализированное взаимодействие между ключевыми модулями инструмента на протяжении всего процесса мониторинга памяти. Каждый модуль выполняет свою роль в обработке событий памяти, начиная с инициализации инструмента пользователем и заканчивая выводом результатов обратно в интерфейс.

Модуль интерфейса служит связующим звеном между пользователем и системой, принимая команды и инициируя процесс мониторинга. Модуль перехвата ответственен за захват событий доступа к памяти, а модуль контекстной атрибуции связывает эти события с конкретными потоками и

модулями. Далее, модуль фильтрации применяет заранее заданные фильтры, чтобы сузить область наблюдения и отфильтровать нерелевантные данные, и, наконец, модуль хранения сохраняет нужную информацию, передавая её обратно в интерфейс для отображения пользователю.

Процесс мониторинга продолжается в цикле до тех пор, пока пользователь не завершит его, что позволяет эффективно отслеживать и анализировать события в реальном времени. Это взаимодействие между модулями делает инструмент гибким и эффективным в решении задач мониторинга памяти.

5.4 Выводы

В данной главе был разработан концептуальный проект архитектуры инструмента мониторинга памяти, опирающийся на требования, сформулированные ранее. Предложенная архитектура имеет модульную организацию и обеспечивает соответствие поставленным целям: каждый модуль отвечает за свою функцию, а совместная работа модулей покрывает все необходимые аспекты (перехват событий, контекстная фильтрация, атрибуция и вывод результатов).

Были получены и представлены три диаграммы, иллюстрирующие различные взгляды на архитектуру инструмента:

– блок-схема алгоритма отражает последовательность операций инструмента, включая альтернативные режимы запуска и цикл обработки событий доступа к памяти;
– диаграмма последовательности показывает обмен сообщениями между пользователем и основными модулями во времени, шаг за шагом.

Эти визуальные модели подтверждают целостность и согласованность архитектуры: все заявленные функциональные блоки интегрированы в единое целое, а их взаимодействие организовано логично и эффективно. Таким образом, концептуальный уровень проектирования завершён – определена структура системы, которая в последующих этапах может быть развёрнута в детальную реализацию и прототип инструмента, удовлетворяющего предъявленным

ЗАКЛЮЧЕНИЕ

Глобальной целью работы является разработка методики и программного инструмента для мониторинга операций чтения/записи/исполнения в память целевого процесса в Linux, с возможностью фильтрации по контексту (поток, модуль, системный вызов, адресный диапазон). В рамках этого этапа НИР предполагалось получение следующих результатов:

1) аналитический обзор и классификация задач мониторинга памяти;
2) сравнительная таблица инструментов по критериям;
3) сформулированные требования к разрабатываемому инструменту;
4) предварительная концепция архитектуры.

Основные результаты работы

В ходе выполнения научно-исследовательской работы по проектированию архитектуры инструмента для контекстного мониторинга доступа к памяти в Linux были получены следующие результаты.

В первой главе проведен комплексный анализ задач мониторинга памяти, включая классификацию основных сценариев использования, определение ключевых параметров наблюдаемых событий (адрес доступа, тип операции, размер, временная характеристика, контекст выполнения) и выявление специфики мониторинга в контексте информационной безопасности. Особое внимание уделено анализу угроз, связанных с ошибками управления памятью (buffer overflow, use-after-free), и техникам перехвата потока исполнения (ROP, JOP).

Во второй главе выполнен детальный сравнительный анализ существующих инструментов и технологий мониторинга памяти, охватывающий три категории решений: аппаратные механизмы (watchpoints и permission faults), механизмы операционной системы (tracepoints/kprobes/uprobes с eBPF, soft-dirty/pagemap, DAMON) и инструменты на уровне приложений (ASan, Valgrind Memcheck, heaptrack, MemGaze). Для каждой категории определены сильные и слабые стороны, профиль накладных расходов и типичные области применения.

В трертьей главе, на основе анализа из предыдущих глав, сформулированы систематизированные требования к разрабатываемому инструменту, разделенные на пользовательские (контекстная фильтрация,

отсутствие необходимости модификации приложения), функциональные (регистрация событий, состав записи о событии, различные виды фильтрации) и эксплуатационные (совместимость с Linux x86_64, приемлемые накладные расходы, надежность работы).

В четверной главе, из полученных функциональных требований, разработана модульная архитектура инструмента, включающая пять основных компонентов: модуль перехвата событий, модуль фильтрации, модуль контекстной атрибуции, модуль хранения результатов и модуль пользовательского интерфейса. Предложена четкая схема взаимодействия между модулями, обеспечивающая эффективную обработку событий и предоставление результатов в удобном формате.

Результаты полностью отвечают поставленной цели и решает все поставленные задачи для этого этапа НИР. Предложенная архитектура предлагает устойчивый фундамент для дальнейшего уточнения и разработки методики контекстного мониторинга.

Направления дальнейших исследований

На основе выполненной работы по проектированию архитектуры инструмента для контекстного мониторинга доступа к памяти в Linux, перспективными направлениями дальнейших исследований являются:

1) разработка методики контекстного мониторинга. Необходимо разработать конкретную методику для получения информации о событиях в памяти и определении его контекста;
2) анализ и выбор технологии реализации. Провести более детальное исследование подходов, описанных в этом этапе НИР и выбрать те их них, которые будут использованы при реализации инструмента;
3) дальнейшая проработка архитектуры разрабатываемого инструмента. Определение интерфейсов модулей инструмента, спецификация API инструмента и определение основных потоков данных в нём.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Президент Российской Федерации. Указ Президента РФ № 400 «О Стратегии национальной безопасности Российской Федерации» [Электронный ресурс]. 2021. URL: http://publication.pravo.gov.ru/Document/View/0001202107020002 (дата обращения: 20.05.2024).
2. Президент Российской Федерации. Доктрина информационной безопасности Российской Федерации (утверждена Указом Президента РФ № 646 от 05.12.2016) [Электронный ресурс]. 2016. URL: http://static.kremlin.ru/media/acts/files/0001201612060002.pdf (дата обращения: 20.05.2024).
3. Государственная Дума Российской Федерации. Федеральный закон от 27.07.2006 № 149-ФЗ «Об информации, информационных технологиях и о защите информации» [Электронный ресурс]. 2006. URL: http://www.consultant.ru/document/cons_doc_LAW_61798/ (дата обращения: 20.05.2024).
4. Monitoring Tools Market (2023–2030): Market Analysis Report. 2023.
5. Chen H., Zhang W. Memory Analysis for Malware Detection: A Comprehensive Survey // Proceedings of the IEEE Security and Privacy Workshops. 2022. Сс. 112–125.
6. Kumar S., Li P. Obfuscated Memory Malware Detection // IEEE Transactions on Dependable and Secure Computing. 2021. Т. 19, № 5. Сс. 3456–3472.
7. Yan C., Li Y. Blindfold: Confidential Memory Management by Untrusted Operating System // Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 2023. Сс. 2105–2119.
8. Kilic O.O. и др. MemGaze: Rapid and Effective Load-Level Memory Trace Analysis // Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER). 2022. Сс. 484–495.
9. Pagani F., Balzarotti D. AutoProfile: Towards Automated Profile Generation for Memory Analysis // ACM Transactions on Privacy and Security. 2022. Т. 25, № 1. С. Article6.
10. Ren J. и др. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory // Proceedings of the 19th European Conference on Computer Systems (EuroSys). 2024.

11. Wang C. и др. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime // Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2022.
12. Solihin Y., Prvulovic M. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging // Proceedings of the IEEE International Symposium on High Performance Computer Architecture. 2021. Сс. 123–136.
13. Arcangeli A., others. Использование userfaultfd для мониторинга доступа к памяти // Linux Kernel Documentation. 2020.
14. Mutlu O., Subramanian L. Research Problems and Opportunities in Memory Systems // Research Problems and Opportunities in Memory Systems. 2014.
15. Liu J. A Study on Modeling and Optimization of Memory Systems // Springer Journal of Software. 2021. Сс. 1–15.
16. Computer Security O.S.T. in. Memory Safety Vulnerabilities and Exploitation [Электронный ресурс]. 2024. URL: https://textbook.cs161.org/memory-safety/vulnerabilities.html.
17. Carnà S. Systemwide Detection and Mitigation of Side-channel Attacks // ACM. 2023.
18. Chen H., Li P., Zhang W. A Survey of Memory Protection Mechanisms for Operating Systems // ACM Computing Surveys. 2020. Т. 53, № 2. Сс. 1–36.
19. Sayas A.S. Comparative Evaluation of Locality Metrics // DIVA Portal. 2025.
20. Buffer Overflow [Электронный ресурс]. URL: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.
21. DWARF Debugging Information Format, Version 5. 2017.
22. Debugging with GDB: Symbol Files and Debugging Information [Электронный ресурс]. 2025. URL: https://www.sourceware.org/gdb/current/onlinedocs/gdb.html/Index-Files.html.
23. Ji X., others. Understanding Object-level Memory Access Patterns Across the Spectrum // ACM Conference Paper. 2017. Сс. 1–10.
24. Linux kernel documentation. BPF Documentation [Электронный ресурс]. 2026. URL: https://docs.kernel.org/bpf/ (дата обращения: 22.01.2026).
25. Gregg B. BPF Performance Tools: Linux System and Application Observability. Addison-Wesley Professional, 2019.
26. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.

27. Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual, Volume 2: System Programming. 2025.
28. Arm Limited. Armv8-A Self-hosted debug. 2020.
29. Kovah X. Intermediate x86 Part 4 [Электронный ресурс]. 2010. URL: https://opensecuritytraining.info/IntermediateX86_files/IntermediateIntelx86-Part4.ppt.pdf (дата обращения: 22.01.2026).
30. Breaker D. Understanding page faults and memory swap-in/outs: when should you worry? [Электронный ресурс]. 2019. URL: https://www.scoutapm.com/blog/understanding-page-faults-and-memory-swap-in-outs-when-should-you-worry (дата обращения: 22.01.2026).
31. Arm Limited. Armv8-A Address translation. 2019.
32. Arm Limited. Learn the architecture: AArch64 Exception Model. 2025.
33. Ltd. A. ARMv8-A Architecture Reference Manual. ARM Holdings, 2017.
34. Using the Linux Kernel Tracepoints — Linux kernel documentation [Электронный ресурс]. URL: https://docs.kernel.org/trace/tracepoints.html (дата обращения: 22.01.2026).
35. Kernel Probes (Kprobes) — Linux kernel documentation [Электронный ресурс]. URL: https://docs.kernel.org/trace/kprobes.html (дата обращения: 22.01.2026).
36. Uprobe-tracer: Uprobe-based Event Tracing — Linux kernel documentation [Электронный ресурс]. URL: https://docs.kernel.org/trace/uprobetracer.html (дата обращения: 22.01.2026).
37. BPF maps.
38. Ariyawangsha V. eBPF programming for Linux Kernel Tracing [Электронный ресурс]. 2022. URL: https://medium.com/@zone24x7_inc/ebpf-programming-for-linux-kernel-tracing-30364dde3fb7 (дата обращения: 22.01.2026).
39. Soft-Dirty PTEs — Linux kernel documentation [Электронный ресурс]. URL: https://docs.kernel.org/admin-guide/mm/soft-dirty.html (дата обращения: 22.01.2026).
40. proc_pid_clear_refs(5) — Linux manual page [Электронный ресурс]. URL: https://man7.org/linux/man-pages/man5/proc_pid_clear_refs.5.html (дата обращения: 22.01.2026).
41. proc(5) — process information pseudo-filesystem.
42. Corbet J. Patching until the COWs come home (part 2) // LWN.net. 2021.

43. Examining Process Page Tables (pagemap) — Linux kernel documentation [Электронный ресурс]. URL: https://docs.kernel.org/admin-guide/mm/pagemap.html (дата обращения: 22.01.2026).
44. proc_pid_pagemap(5) — Linux manual page.
45. proc_pid_maps(5) — Linux manual page.
46. proc_pid_smaps(5) — Linux manual page.
47. Memory changes tracking [Электронный ресурс]. 2019. URL: https://criu.org/index.php?title=Memory_changes_tracking (дата обращения: 22.01.2026).
48. DAMON: Data Access MONitoring — Linux kernel documentation [Электронный ресурс]. URL: https://docs.kernel.org/mm/damon/index.html (дата обращения: 22.01.2026).
49. Serebryany K. и др. AddressSanitizer: A Fast Address Sanity Checker // USENIX Annual Technical Conference (ATC). 2012.
50. Nethercote N., Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation // Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2007.
51. Valgrind User Manual. 2025.
52. heaptrack: A heap memory profiler for Linux [Электронный ресурс]. URL: https://github.com/KDE/heaptrack (дата обращения: 22.01.2026).
53. Kilic O.O. и др. MemGaze: Rapid and Effective Load-Level Memory Trace Analysis // 2022 IEEE International Conference on Cluster Computing (CLUSTER). 2022. Сс. 484–495.