Защищенная среда отладки и исполнения критических приложений во встраиваемых системах.

Промежуточный отчет по работе «Защищенный режим — 2»
за период с 15.12.2017 по 30.07.2018

Мустафин Т. Р., Алехин А. И., Малахов И. Ю., Мареев Е. Г., Федоров А. В., Лубинец М. И., Макаев Б. О., Кравцунов Е. М.

АО «МЦСТ», Москва, 2 августа 2018 г.

Содержание

1. Уязвимости языка Си.

2. Примеры атак, использующих уязвимости. ЗР в действии.

3. Применение в реальной задаче: работа тестов стандарта IEC 61850 сетей и систем связи на подстанциях в ЗР.

4. Продвижение в развитии защищенного режима:

   4.1 Развитие поддержки ЗР в компиляторе **rel-23-0**.

   4.2. Перевод uclibc-ng на компилятор **rel-23-0**, поддержка в динамическом линковщике.

   4.3. Поддержка отладчика gdb в динамическом линковщике.

   4.4. Механизм заказов и освобождения памяти для ЗР, реализация в библиотеке uclibc-ng, поддержка в ядре linux 4.9.

   4.5. Continious Iintegration для библиотек ЗР.

1. Уязвимости языка Си
1.1 Обращение к несуществующему элементу массива

```
char buf[10];
buf[10] = 'A';
```

Для ЗРИ, дескриптор buf во время выполнения buf[10] = 'A':

eTag (8)	iTag (3)	RW (2)	Base (59)	Size (32)	Index (32)
0xfc	0b100	0xb11	0x********	0xa	0xa

Index >= Size => exc_illegal_operand

1. Уязвимости языка Си
1.2 Использование указателей после неправильных арифметических операций над ними

```
int ptr[3] = {1, 2, 3};

int ptr1[3] = {-1, -2, -3};

int * a;

a = ptr;

a -= 3;

printf("*a = %d\n", *a);
```

Для ЗРИ, дескриптор a во время выполнения a*:

eTag (8)	iTag (3)	RW (2)	Base (59)	Size (32)	Index (32)
0xfc	0b100	0xb11	0x********	0x3	0xfffffffd

Index >= Size => exc_illegal_operand

1. Уязвимости языка Си
1.3 Использование неинициализированных значений переменных

```
int foo;
if (foo==0)
        printf("foo == 0\n");
else
        printf("foo != 0\n");
```

Для ЗРИ, переменная foo во время выполнения foo == 0:

eTag (2)	Value (32)
0b11	0x********

eTag == DT (Diagnostic Type, 0b11) => exc_illegal_operand

1. Уязвимости языка Си
1.4 Использование зависших указателей после освобождения памяти

```
int* a = malloc(10 * sizeof(int));

free(a);

int * b = malloc(10 * sizeof(int));

for (i = 0; i < 10; i++) b[i] = 1;

for (i = 0; i < 10; i++) a[i] = 2;
```

В защищенном режиме освобожденная область памяти не перевыделяется пользователю, до тех пор, пока операционная система не удалит все копии дескрипторов, описывающие эту область памяти.

Незащищенный режим:	Защищенный режим:
b[0...9] = 2;	b[0...9] = 1;
	Либо SIGILL на a[i] = 2, т. к. eTags==NV

1. Уязвимости языка Си
1.5 Утечки памяти

```
while((buffer=malloc(1024*1024)) != NULL && mb != max) {
    memset(buffer, 0, 1024*1024);
    mb++;
}
```

Утечки памяти в операционной системе Linux контролируются с помощью процесса Out Of Memory Killer. В защищенном режиме утечки памяти особенным способом не контролируются.

1. Уязвимости языка Си

1.6 Использование неправильных типов и/или неверного количества переменных в вызовах функций с переменным числом аргументов

```
int foo(int a, …) {
        va_list p;
        va_start(p, a);
        int b = va_arg(p, int);
        va_end(p);
        return a + b;
}
int main() {
        int a = 0;
        return foo(a);
}
```

- Защищенным режимом в общем случае не контролируется.
- Контроль частных случаев возможен из-за проверки внешних тегов.

ЗРИ, переменная a во время подсчета a+b:

eTag (2)	Value (32)
0b11	0x********

eTag == DT (Diagnostic Type, 0b11) => exc_illegal_operand

2. Примеры атак, использующих уязвимости
2.1 Переполнение буфера

```
void func1(){ printf("func1() was called\n"); }
void func2(){ printf("func2() was called\n"); }
int main(int argc, char **argv, char **envp) {
        void (*functionpointer)(void) = func1;
        char buffer[50];
        memcpy(buffer, argv[1], strlen(argv[1]));
        functionpointer();
        return 0;
}
```

- Слишком большой массив данных на входе программы, приводит к изменению указателя на функцию
- В частности, вместо функции func1() может быть вызвана функция fucn2().

Для ЗР, дескриптор buffer в memcpy() во время записи 50-го байта:

eTag (8)	iTag (3)	RW (2)	Base (59)	Size (32)	Index (32)
0xfc	0b100	0xb11	0x********	0x32	0x32

Index >= Size => exc_illegal_operand

2. Примеры атак, использующих уязвимости
2.2 Уязвимость spectre

```
char a1[10]="0123456789";
char secret[10]="secret";
char* a2;
char func(int offset) {
  if (offset<10)
    a2[a1[offset]*64];
}
```

- Вызов функции func() с offset > 10, может привести к спекулятивному чтению a1[x] и a2[a1[x]*64].
- Из-за спекулятивного чтения в кэш загружаются элементы массива a2 с адресами, соответствующими значениям байт секретной строки.
- Анализируя время доступа к кэшу, можно узнать секретную строку.

Для ЗРИ, дескриптор a1 во время спекулятивного чтения a1[offset], при offset > 10:

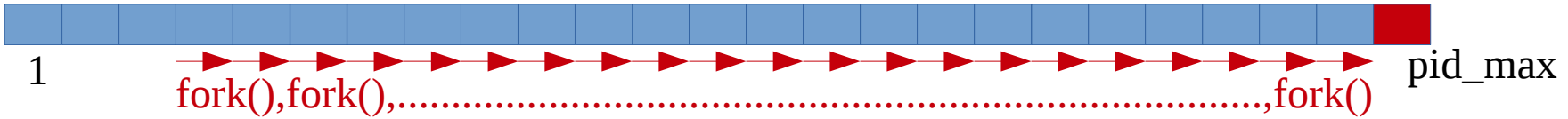
eTag (8)	iTag (3)	RW (2)	Base (59)	Size (32)	Index (32)
0xfc	0b000	0xb11	0x********	0xa	offset

sm, Index >= Size => операция чтения завершается с результатом диагностического типа, eTag=DT. Данные в кэш не загружаются.

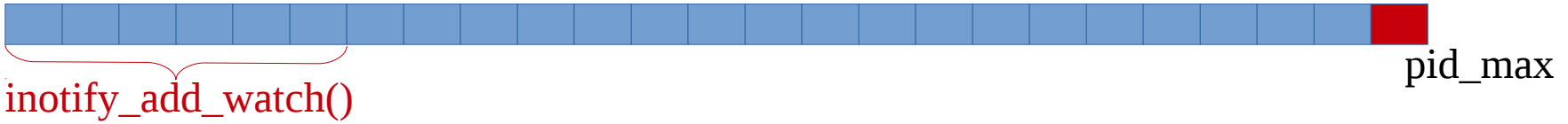
2. Примеры атак, использующих уязвимости
2.3 Скрытие непривилегированного процесса

Процесс можно спрятать от утилит procps с помощью fork() и inotify_add_watch().

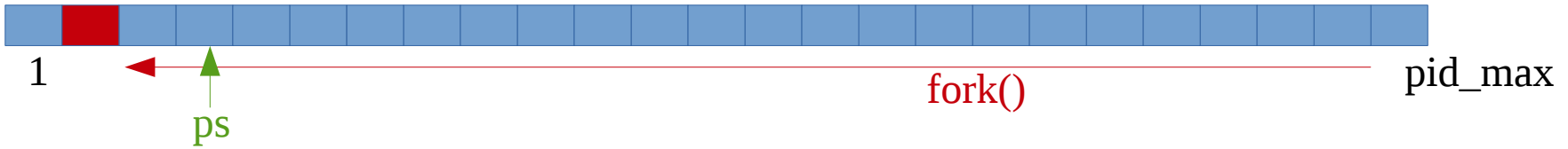
1) fork() пока c_pid > p_pid; exit() в min(c_pid, p_pid)



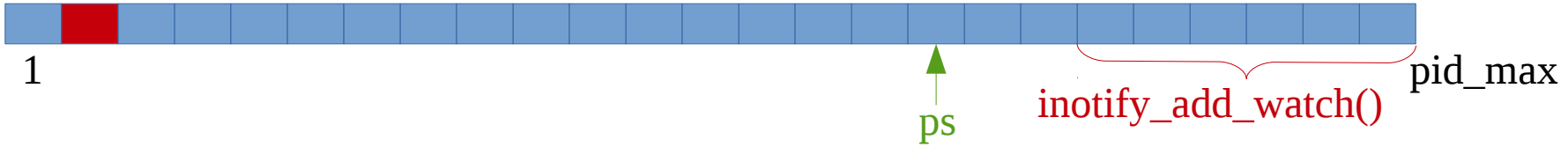
2) inotify_add_watch(IN_OPEN, /proc)



3) При срабатывании монитора, fork() пока c_pid < p_pid; exit() в p_pid



4) inotify_add_watch(IN_CLOSE_NOWRITE, /proc).



5) При срабатывании монитора, вернуться к 1).

Алгоритм сокрытия процесса основан на системных вызовах ядра Linux, поэтому работает на x86, e2k и в защищенном режиме.

3. Применение в реальной задаче
Библиотека libiec61850

Ключевое слово «Цифровая подстанция»: сигналы от современного оконечного оборудования передаются через Ethernet на промышленный компьютер, где производится анализ формы сигнала. Такой анализ — сложная вычислительная задача: БПФ → Комплексное преобразование → Сравнение.

SV — протокол передачи данных для обработки.

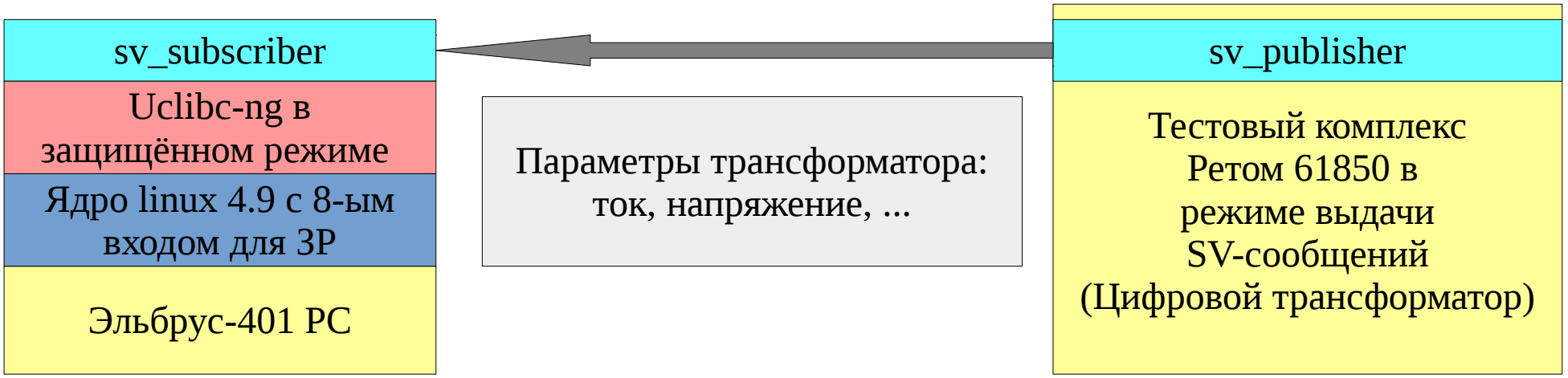
Искажение сигнала — причина для отключения линии по протоколу GOOSE.

Для взаимодействия с другими промышленными компьютерами используется протокол MMS.

Идеология — ABB, Alstom, Siemens, открытая реализация - libiec61850:
http://libiec61850.com/libiec61850/

3. Применение в реальной задаче
Производительность в ЗР на тесте **sv_subscriber**

sv_subscriber: приём параметров от цифрового трансформатора. Сервер - РЕТОМ 61850. Клиент — машина Эльбрус-401 РС.

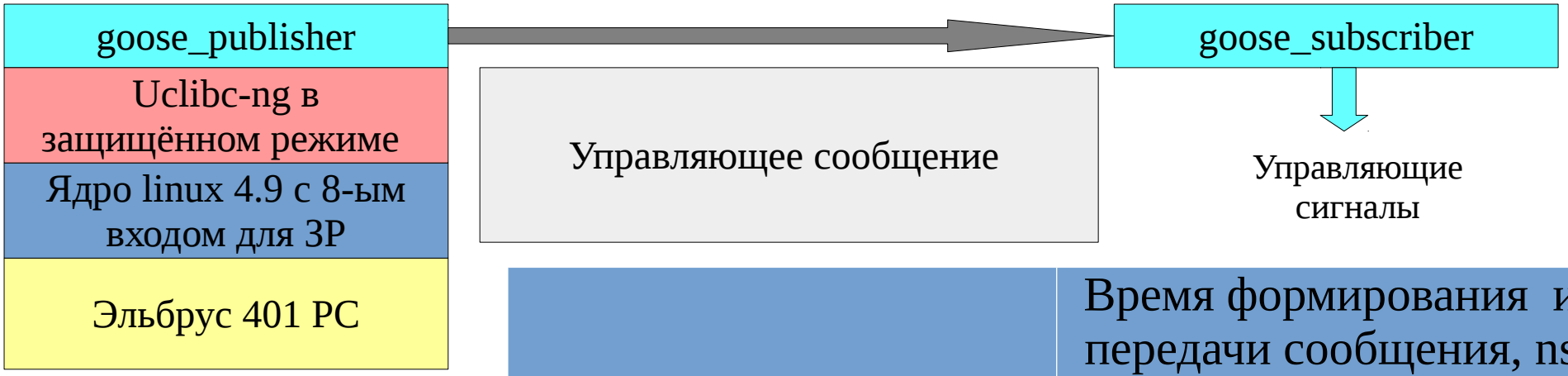


	Время итерации обработки полученных данных, ns
Обычный режим	1665
Защищённый режим	2011

ЗР – 21 % издержки

3. Применение в реальной задаче
Производительность в ЗР на тесте **goose_publisher**

goose_publisher: формирование и отправка goose-сообщений для управляющих элементов (цифровое реле). Прием сообщений осуществляет тест **goose_subscriber**, моделирующий реле на машине x86.

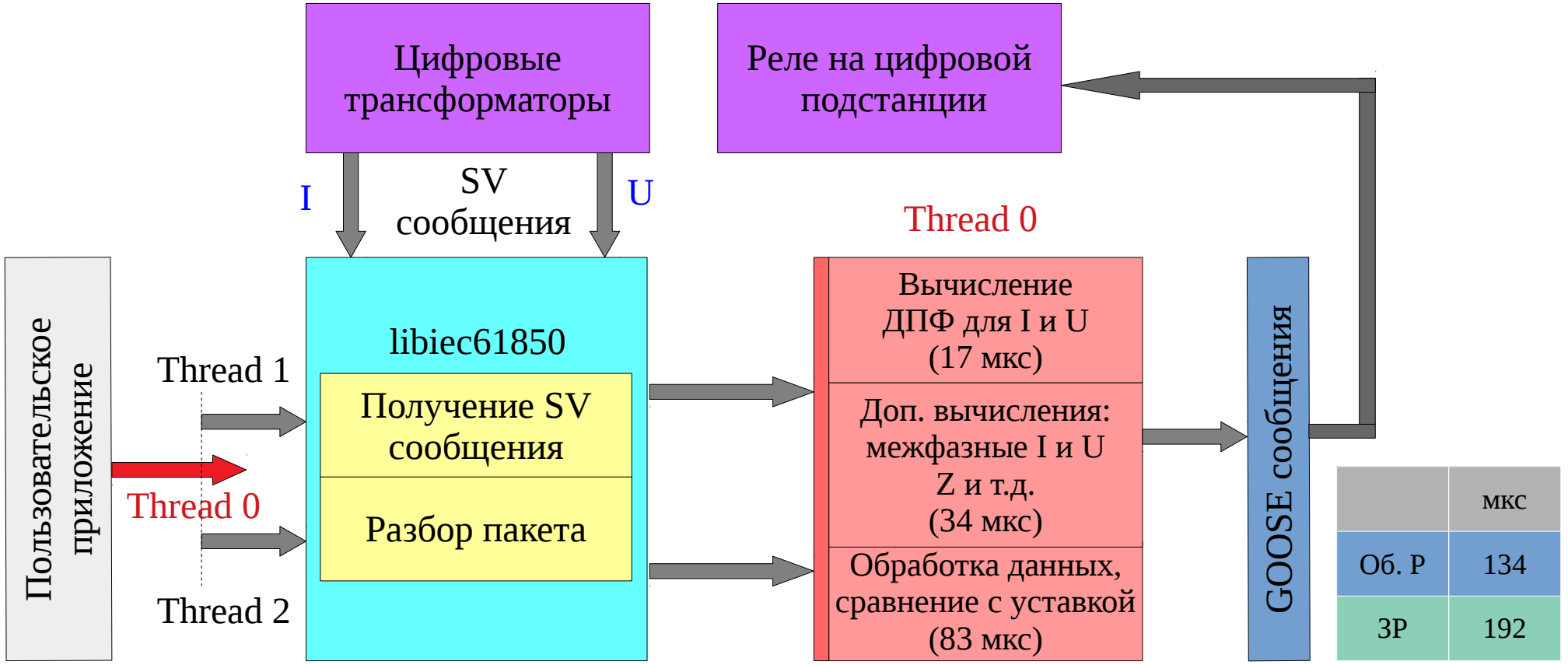


ЗР – 64 % издержки

	Время формирования и передачи сообщения, ns
Обычный режим	16142
Защищённый режим	26514

3. Применение в реальной задаче
Пример работы libiec61850 в схеме релейной защиты автоматики

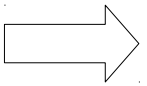
- Прием о обработка данных происходит в 2 потока: Thread 1 обменивается сообщениями с трансформатором тока I, Thread 2 — с трансформатором напряжения U.
- После фазы вычисления и сравнения с пороговым значением принимается решение об отправке GOOSE сообщения.



3. Применение в реальной задаче
Исправления в коде libiec при переносе в 3P

- Не использовать атрибут «packed» для структур:

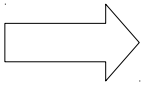
```
struct sMmsVariableSpecification
__attribute__ ((__packed__))
{
    uint8_t size;
    uint8_t maxSize;
    uint8_t* octets;
    ……
```



```
struct sMmsVariableSpecification
{
    uint8_t size;
    uint8_t maxSize;
    uint8_t* octets;
    ……
```

- Не допускать сравнения дескрипторов с отрицательными значениями указателей текущих позиций:

```
for(p = buf + sizeof(buf) - 1; p >=
buf; p--, native >>= 8)
……
```



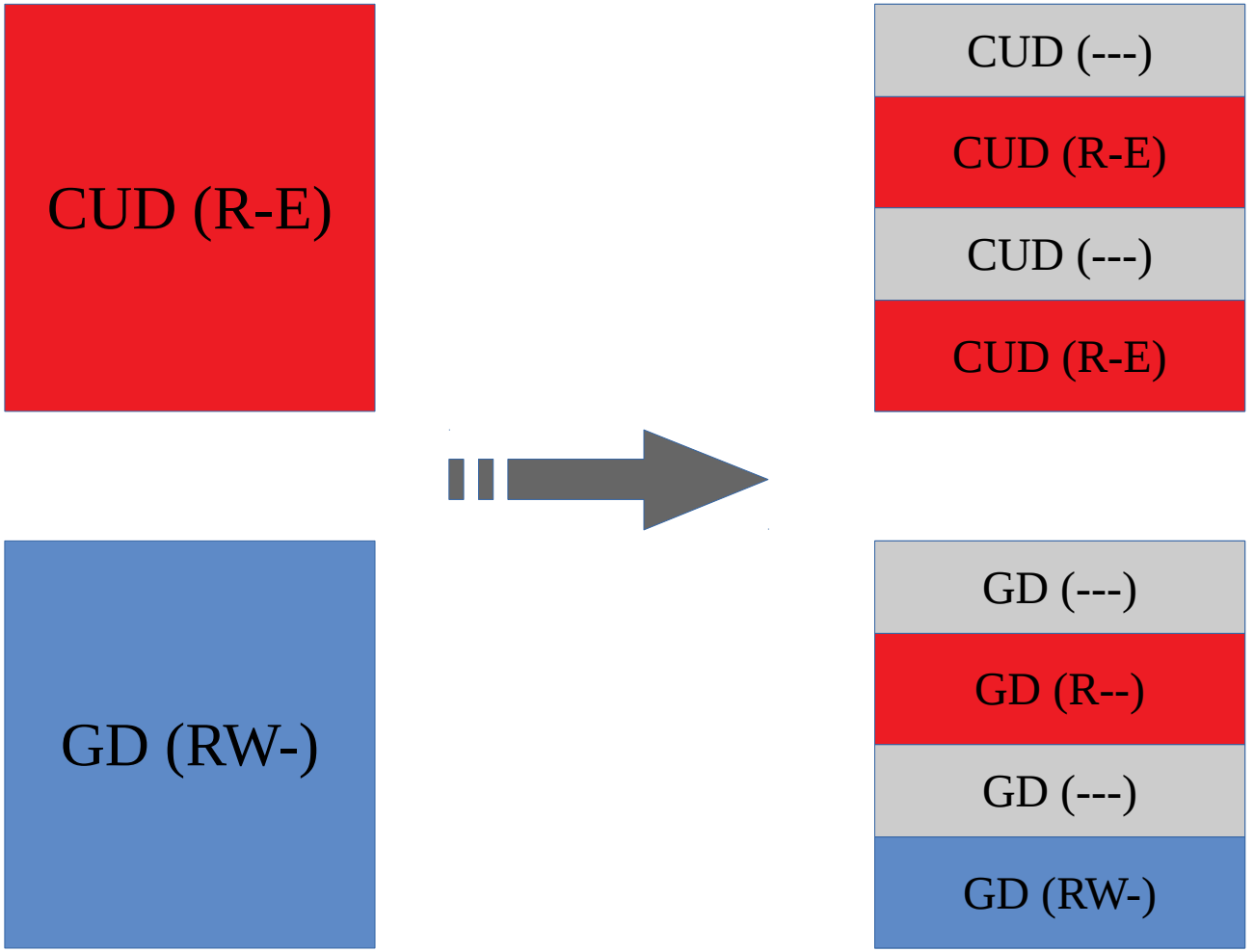
```
for(p = buf + sizeof(buf) - 1;
p+1000 >= buf+1000; p--, native
>>= 8)
…….
```

- Был обнаружен выход за границы выделенной памяти, не проявлявшийся в обычном режиме. Патч прнят разработчиками libiec.

```
size = self->value.octetString.maxSize;
memcpy(self->value.octetString.buf, update->value.octetString.buf, size);
```

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

Отображение произвольного числа ELF сегментов
в сегменты кода и данных защищенной программы



4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

- Размещение read-only, включая служебные данные программы, и read-write данных в сегменте данных (GD).
- Поддержка “read-only after relocation” секций ELF.
- Симметричное построение обращений к read-only и модифицируемым данным.
- Размещение секций кода и данных по заданным смещениям относительно CUD.base и GD.base.
- Сокращение числа случаев, в которых требуется адаптация скриптов линковщика к защищенному режиму.

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

Унификация защищенного и обычных режимов в СП для rel-23-0

- Построение кода, выполняющего статическую инициализацию указателей перенесено с компилятора на линкер.
- Переход с устаревшего init/fini механизма вызова языковых конструкторов и деструкторов на {init,fini}_array.
- Реализация copy-релокейшенов и выполнение динамических вызовов через PLT позволяет собирать защищенные динамические исполняемые файлы без -fPIC.

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

- Исправлена реализация weak undefined символов. За счет построения обращений к ним через GOT удалось добиться равенства указателя на такой символ NULL Pointer при отсутствии определения.
- Переход от кустарной libgcc на регулярную версию от gcc-5.5.0, используемую в обычных режимах.
- Интерфейс между динамической защищенной программой и gdb реализован по аналогии с используемым в glibc для обычных режимов.

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

Прочие изменения в реализации защищенном режима для rel-23-0

- Реализована совместимость с Elbrus V6 путем перехода на 128 битную PL в СП и интерфейсе динамического загрузчика с ядром.
- За счет доработок в статической инициализации указателей из интерфейса с ядром исключена передача избыточной информации, которую может предоставить сам модуль.
- Исправлены ошибки при передаче ядром аргументов, позволяющих библиотеке различать static и dynamic случаи, стартовой функции защищенной программы.

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

- В загрузчик защищенных ELF’ов в ядре добавлено создание недостающих записей auxv[AT_{PHNUM,PHDR,ENTRY}] с учетом изменений в размещении секций данных в runtime.

4. Продвижение в развитии защищенного режима: 4.2. Перевод uclibc-ng на компилятор **rel-23-0**, поддержка в динамическом линковщике.

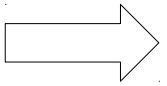
1. Uclibc-ng адаптирована для сборки компилятором **rel-23-0**
2. Система сборки uclibc-ng доработана для одновременной сборки как в обычном так и в защищённом режиме
3. Uclibc-ng в защищённом режиме внедрена в систему сборки компилятора **rel-23-0**

4. Продвижение в развитии защищенного режима: 4.2. Перевод uclibc-ng на компилятор **rel-23-0**, поддержка в динамическом линковщике.

Для загрузки модулей в память в защищённом режиме используется системный вызов uselib, который возвращает указатель на структуру типа mdd_t с необходимой информацией о загруженном модуле

Предыдущие версии

```
struct mdd_t {
    void (* mdd_init_got) ();
    void (* mdd_init) ();
    void (* mdd_fini) ();
    void (* mdd_start) (int, ...);
    void *mdd_got;
}
```



rel-23-0

```
struct mdd_t {
    void (* selfinit) ();
    void *gd;
}
```

- Вызов mdd_init_got для инициализации GOT
- Вызов mdd_init для инициализации глобалов и вызова конструкторов
- Использование mdd_got для дальнейшего доступа к GOT
- mdd_start — точка входа в основную программу

- Вызов selfinit для инициализации GOT и глобалов
- Использование дескриптора области данных gd для дальнейшего доступа к данным и GOT
- Получение дескриптора на точку входа в программу через GOT

4. Продвижение в развитии защищенного режима:
4.2. Перевод uclibc-ng на компилятор **rel-23-0**, поддержка в динамическом линковщике.

- Добавлена поддержка релокаций, появившихся в компиляторе **rel-23-0**:

  - R_E2K_AP (на данные)

  - R_E2K_PL, R_E2K_32_JMP_SLOT (на функции)

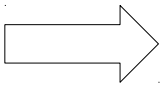
  - R_E2K_32_COPY

- Добавлена поддержка WEAK-связывания, появившегося в компиляторе **rel-23-0**

4. Продвижение в развитии защищенного режима:
4.3. Поддержка отладчика gdb в динамическом линковщике.

Предыдущие версии

```
struct elf_resolve {
    Elf32_Addr l_data_addr;
    Elf32_Addr l_code_addr;
    unsigned int l_name;
    unsigned int l_next;
    unsigned int l_prev;
……
```



rel-23-0

```
struct elf_resolve {
    Elf32_Addr l_data_addr;
    Elf32_Addr l_code_addr;
    char *libname;
    ElfW(Dyn) *dynamic_addr;
    struct elf_resolve *next;
    struct elf_resolve *prev;
……
```

В **rel-23-0** версии компилятора gdb умеет работать с защищёнными дескрипторами в структурах, хранящих информацию о загруженных модулях. Поэтому информация о загруженных модулях хранится в виде максимально приближенным к обычному режиму.

4. Продвижение в развитии защищенного режима:
4.4. Механизм заказов и освобождения памяти для ЗР

Требования к **malloc**() **- free**() для ЗР, баг 93263:

- Malloc заказывает память у ОС большими порциями, и выдает приложению кусками нужных размеров. Для этого маленькие куски одного размера хранятся в пулах - все как в обычном незащищенном malloc, только указатель на пользовательский кусок нужно будет делать конверсией из большого, с уменьшением поля size.

- Free, в отличие от незащищенного, не освобождает память, но делает пометку и заносит указатель в список list_Descriptors_to free.

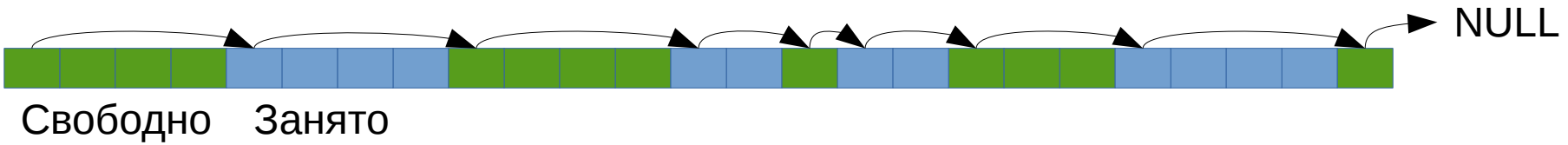
- Если размер списка list_Descriptors_to free превысил порог, либо если память закончилась при попытке заказа, делается системный вызов, в который передается список для освобождения. При этом ОС проходит всю память, стеки и регистры и уничтожает залипшие значения указателей.

4. Продвижение в развитии защищенного режима:
4.4. Механизм заказов и освобождения памяти для ЗР

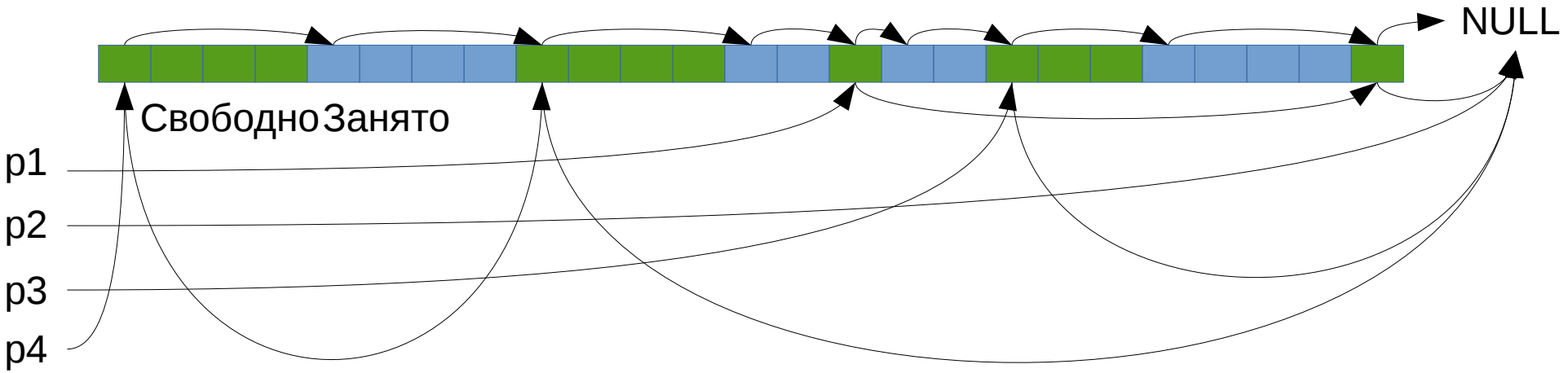
Выбор malloc в качестве основы для ЗРИ

Требование: «Malloc заказывает память у ОС большими порциями, и выдает приложению кусками нужных размеров. Для этого маленькие куски одного размера хранятся в пулах - все как в обычном незащищенном malloc.»

- ✗ malloc-simple: malloc() = mmap(), free() = munmap(). Заказывает память нужного размера и передает пользователю.
- ✗ malloc: заказывает память большими порциями, все куски памяти хранятся в едином списке.



- ✓ malloc-standard: dlmalloc, заказывает память большими порциями, все куски памяти хранятся в едином списке. Освобожденные области сортируются в списки (пулы) по размерам.

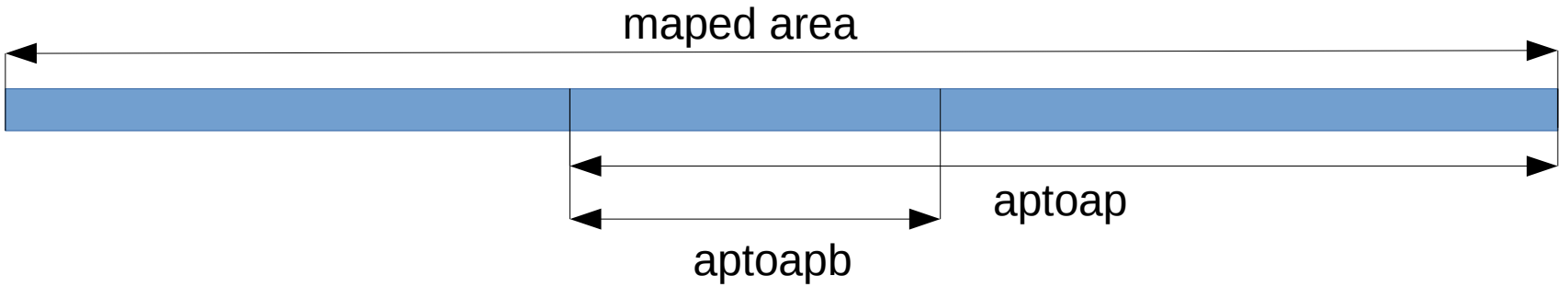


4. Продвижение в развитии защищенного режима:
4.4. Механизм заказов и освобождения памяти для ЗР

Функции mem2pmem и pmem2mem

«Указатель на пользовательский кусок нужно делать конверсией из большого, с уменьшением поля size.»

- **mem2pmem** уменьшает размер описываемой дескриптором области



- Для ЗР все маппированые области составляются в список. **pmem2mem** по обрезанному дескриптору ищет исходную область в списке всех маппированых областей. С дескриптором исходной области работает основной код free/malloc.


```
find i: mapped_area_i_start <= mem < mapped_area_i_end;
mem = mapped_area_i_start – (ulong) mapped_area_i_start + (ulong) mem;
free_internal(mem);
```

4. Продвижение в развитии защищенного режима:
4.4. Механизм заказов и освобождения памяти для ЗР

Функция **free**()

- **free**, в отличие от незащищенного, не освобождает память, но делает пометку и заносит указатель в список list_Descriptors_to free.
- Если размер списка list_Descriptors_to free превысил порог, либо если память закончилась при попытке заказа, делается системный вызов, в который передается список для освобождения.

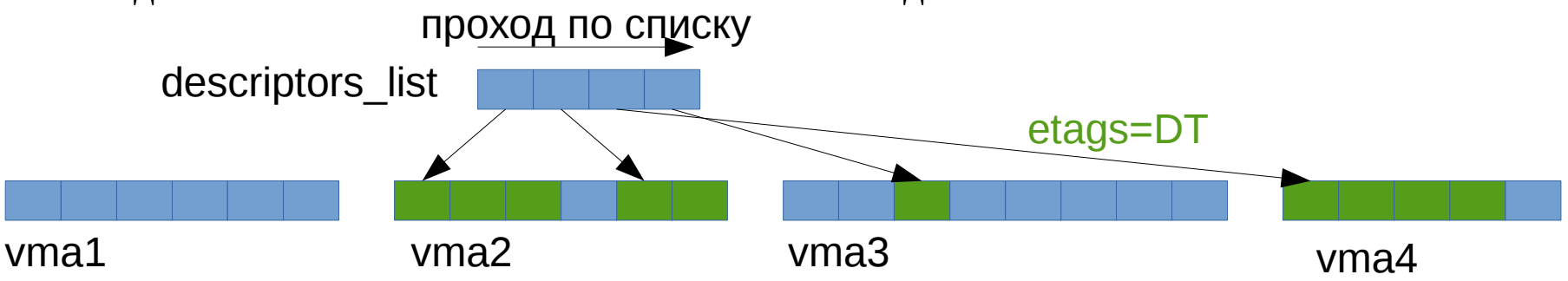
```
void free(void* mem)
{
     __MALLOC_LOCK;
    (av->list_descriptors_to_free)[av->list_descriptors_to_free_size] = mem;
    av->list_descriptors_to_free_size++;

    if (av->list_descriptors_to_free_size >=
         av->list_descriptors_to_free_size_max) {
        clean_descriptors(av->list_descriptors_to_free,
                          av->list_descriptors_to_free_size);
        for (i = 0; i < av->list_descriptors_to_free_size; i++) {
            mem = (av->list_descriptors_to_free)[i];
            mem = pmem2mem(mem, av);
            free_internal(mem);   /* free() function from malloc-standard */
        };
        av->list_descriptors_to_free_size = 0;
    };
     __MALLOC_UNLOCK;
}
```

4. Продвижение в развитии защищенного режима:
4.4. Механизм заказов и освобождения памяти для ЗР

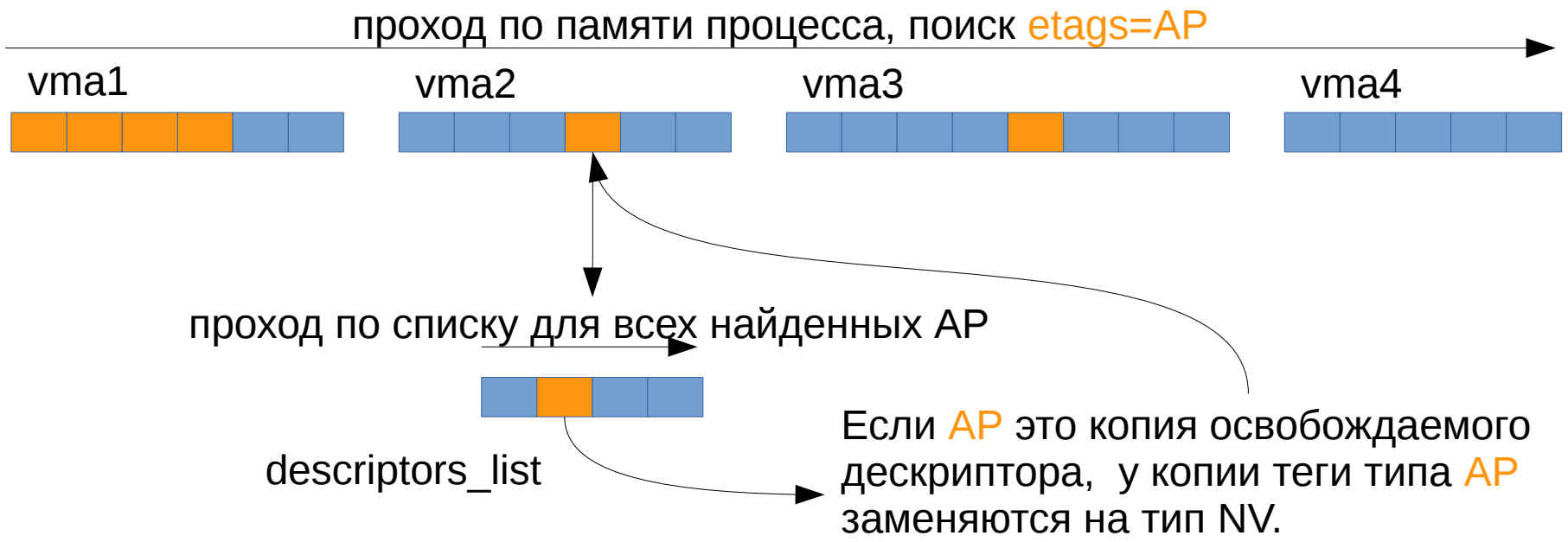
Системный вызов clean_descriptors

ОС проходит всю память, стеки и регистры и уничтожает залипшие значения указателей.

1. Освобождаемые области записываются значениями диагностического типа



2. Из памяти удаляются все копии освобождаемых дескрипторов

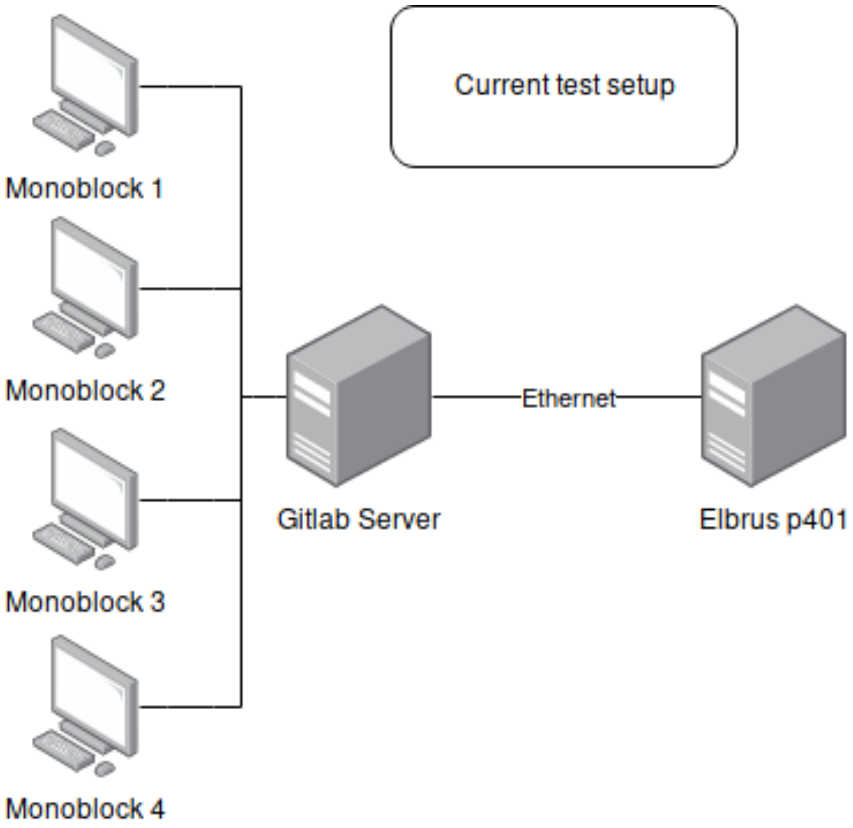


4. Продвижение в развитии защищенного режима:
4.5. Continious Iintegration (CI) для библиотек ЗР.

1. CI для защищенного входа в ядро linux 4.9 (вход № 8)
2. CI для uclibc-ng в ЗР
3. Ускорение кросс-компиляции с помощью Distcc и Ccache

4. Продвижение в развитии защищенного режима:
4.5. Continious Iintegration (CI) для библиотек ЗР.

CI: вход №8 в ядро linux

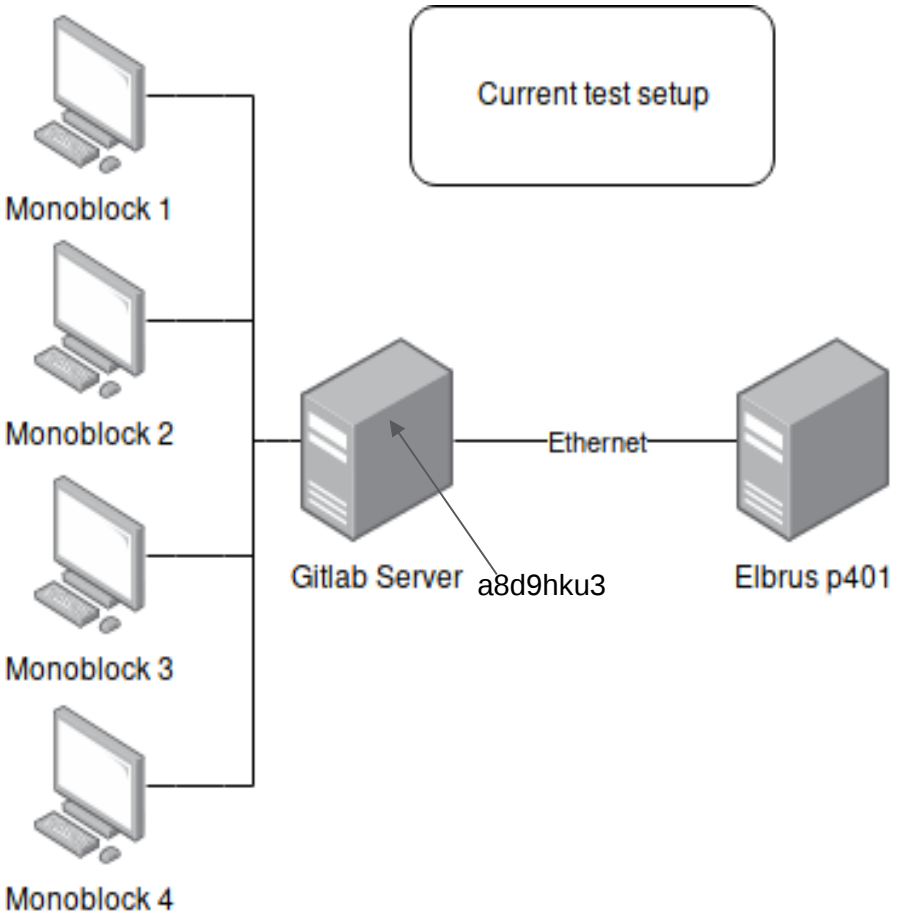


Merge/Push изменений в ветку с CI скриптом приводит к выполнению:

1. Сборка образа ядра linux
2. Раздача образа через AoE (ATA over Ethernet)
3. Проверка корректного запуска машины под новым ядром
4. Сохранение логов с информацией о boot’е и логов e2k машины
5. Запуск тестов на вход №8 в ядро – тесты uclibc-ng-test в ЗРИ
6. Сохранение логов тестов

4. Продвижение в развитии защищенного режима:
4.5. Continious Iintegration (CI) для библиотек ЗР.

CI для uclibc-ng

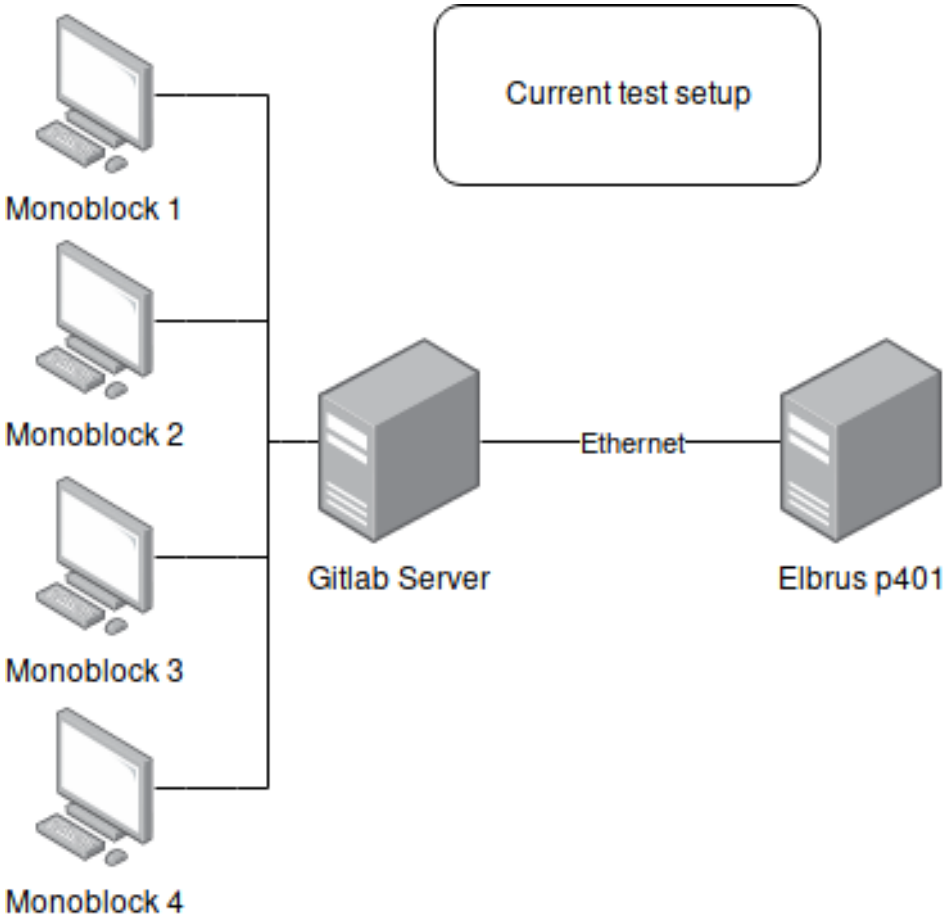


Коммит изменений в репозиторий GitLab uclibc-ng запускает последовательность действий:

1. Собирается тулчейн с новой версией uclibc-ng в докер-контейнере
2. Запускается сборка uclibc-ng и тестов в докер-контейнере
3. Собирается образ buildroot для тестирования uclibc-ng
4. Образ передается на Elbrus p401
5. Тестирование запускается внутри образа buildroot при помощи chroot/LXC контейнера

4. Продвижение в развитии защищенного режима:
4.5. Continious Iintegration (CI) для библиотек ЗР.

CI для uclibc-ng в ЗР



После тестирования uclibc в незащищенном режиме запускается последовательность для ЗРИ:

1. Переиспользуется тулчейн, собранный для обычной версии
2. Сборка библиотеки и тестов в ЗР
3. Запуск происходит на отдельной машине с последней версией ttable8 ядра

4. Продвижение в развитии защищенного режима:
4.5. Continious Iintegration (CI) для библиотек ЗР.

Distcc, Ccache: ускорение компиляции для CI

Distcc - программа, позволяющая распределить задачи компиляции между указанными удаленными машинами

Ccache - программа, сохраняющая кэш компиляции, что позволяет ускорить рекомпиляцию исходного кода

Результаты ускорения компиляции ядра линукс 4.9:

	Последовательно (минуты)	Распараллеливание Distcc (минуты)	Распараллеливание Distcc+Ccache (минуты)
Linux kernel 4.9	22	16	3

Что дальше:

- Создание прототипа промышленного компьютера на Эльбрус-4С (мезонин COM-Express с Эльбрус-4С 800 МГц + носитель + мезонин Radeon + промышленная конструкция, без вентилятора).

ПО — Эльбрус-Д с защищенным контейнером.

Изделие двойного назначения (проект «Тишина»*).

- Расширение списка реализованных CVE, не работающих в защищенном режиме.
- Расширение набора библиотек для защищенного контейнера
- Реализация инфраструктуры отладки в защищенном режиме, доступной внешнему миру (проект «Площадка»*).

* Тишина и Площадка — придуманные нами шифры.

Спасибо за внимание

Backup

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

1. В линковщике реализована поддержка отображения произвольного количества сегментов ELF-файлов как в сегмент кода, так и в сегмент данных защищенной программы. Поскольку каждый сегмент ELF-файла имеет свои разрешения на чтение/запись/исполнение, это позволяет использовать сегмент данных защищенной программы для размещения как read-only, так и read-write данных за счет установки подходящих битов защиты соответствующих страниц при загрузке. Также это позволяет поддержать в защищенном режиме концепцию "read-only after relocation" данных, доступных для записи лишь на время выполнения перемещений.
2. Реализован патч для поддержки изменений из п.1 в загрузчике ELF-файлов в ядре.
3. Реализация следующей функциональности в компиляторах защищенного режима максимально унифицирована с обычными.
4. Со стороны системы программирования реализована совместимость с 128-битным PL в elbrus-v6. Предложены соответствующие изменения в интерфейсе с ядром.
5. Попутно с реализацией п.4 из интерфейса была исключена передача избыточной информации, которую загружаемый модуль может предоставить самостоятельно без участия ядра, и исправлены ошибки при передаче параметров стартовой функции защищенной программы.
6. Интерфейс между защищенной программой и gdb реализован максимально близко к используемому в glibc.

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

1. В линковщике реализована поддержка отображения произвольного количества сегментов ELF-файлов как в сегмент кода, так и в сегмент данных защищенной программы. Поскольку каждый сегмент ELF-файла имеет с… сегмент данных защищен… за счет установки подходящ… ляет поддержать в защищен… иси лишь на время выполне…
2. Реал…
3. Реали… кима максимально унифици…
4. Со ст… им PL в elbrus-v6. Предлож…
5. Попут… формации, которую загружае… злены ошибки при передаче…
6. Интер… к используемому в glibc.

Другие преимущества по сравнению со старой схемой:

a. Обращения к read-only и read-write данным в компиляторе строятся симметричным образом. Размещение read-only данных в сегменте кода никогда в полной мере не поддерживалось из-за необходимости обращаться к ним иным образом, к тому же оно приводит к неоправданным конфликтам, если данные, реально используемые только для чтения, в одном модуле объявлены как read-only, а в другом - как read-write.

b. Появляется возможность размещать отдельные секции кода и данных по заданным смещениям относительно баз соответствующих сегментов. В ряде случаев это упрощает отслеживание исполнения определенных участков кода и обращения к определенным секциям данных при отладке оптимизирующего компилятора, поскольку их смещения перестают зависеть от изменения размеров других фрагментов кода и данных.

c. Значительно сокращается количество случаев, в которых требуется адаптация скриптов линковщика к защищенному режиму.

4. Продвижение в развитии защищенного режима:
4.1 Развитие поддержки ЗР в компиляторе rel-23-0.

1. В линк
как в сегме
имеет сво
защищенно
подходящи
защищенно
выполнени
2. Реализ
3. Реализа
унифициро
4. Со стор
Предложен
5. Попутно
загружаем
передаче па
6. Интерфе
glibc.

a. Построение кода, отвечающего за статическую инициализацию указателей, перенесено с компилятора на линкер, размещающий его в единой функции самоинициализации модуля __selfinit. Это позволило избавиться от поддержки нестандартных секций инициализации `.init.*, а также решило проблему статической инициализации указателем на массив, имеющий неопределенный размер на момент компиляции.

b. Произведен переход с устаревшего механизма вызова языковых конструкторов и деструкторов через специальные функции `_{init,fini} ()' склеиваемые из фрагментов в различных модулях компиляции, на механизм `.{init,fini}_ array', использующий указатели на конструкторы/деструкторы, размещаемые в соответствующих секциях. Помимо большей простоты, новый механизм позволяет более строго релизовать поддержку приоритетов конструкторов и деструкторов с помощью `_attribute__ ((init_priority (xxx)))'.

c. Благодаря реализации в защищенном режиме copy-релокейшенов и выполнению динамических вызовов по аналогии с обычными режимами через трамплин (PLT) удалось избавиться от давнишнего ограничения на сборку динамических исполняемых файлов с `-fPIC'. Это должно существенно сэкономить затраты на адаптацию к защищенному режиму систем конфигурации и сборки реального софта.

d. Исправлена поддержка неопределенных символов со слабым связыванием (weak undefined symbols) за счет построения обращений к ним через GOT при любых опциях компиляции. Теперь по аналогии с обычными режимами указатель на такой символ будет равен Null Pointer при отсутствии определения в других модулях.

e. Перечисленные выше изменения позволили отказаться от использования в защищенном режиме кустарной реализации libgcc и перейти на использование регулярной версии от gcc-5.5.0 как и в обычных режимах.