Buildography v3.7.5

Описание ПО и руководство пользователя

2026-06-03

Содержание

1. Общие сведения

Buildography — инструмент для идентификации реквизитов сборки: файлов программ на входных языках, зависимостей (системных библиотек, заголовочных файлов и т.п.), трансляторов и их конфигурационных файлов.

Предполагаемый сценарий применения — получение структурированной информации о выполнении сценария сборки программного обеспечения в целях отладки, инспектирования или аудита. Buildography предполагается использовать в качестве инструментального средства для обеспечения контроля сборочного процесса на уровне файлов и процессов операционных систем. Контроль сборочного процесса подразумевает, среди прочего, что сборка должна выполняться с использованием

исключительно разрешённых файлов. Разрешённые файлы — это заранее указанные исходные файлы программы, а также файлы, создаваемые в процессе сборки с использованием только разрешённых файлов.

Инструмент позволяет осуществить при сборке ПО контроль использования:

- разрешённых файлов исходного кода;
- предсобранных бинарных файлов;
- файлов, не являющихся разрешёнными файлами исходного кода и не создаваемых в результате выполнения команд сборки.

При этом инструмент может быть использован для трассировки произвольных процессов в Linux.

Buildography отслеживает системные вызовы дочернего дерева процессов с помощью системного вызова ptrace и собирает следующую информацию:

- Рабочая директория;
- Аргументы командной строки порождаемых процессов;
- Данные о файлах, которые открывались этими процессами;
- Информация, идентифицирующая вызываемые программы.

Инструмент записывает её в выходной файл в формате JSON.

Этот файл может быть использован для анализа происхождения файлов, возникших в результате сборки: из каких исходных файлов и с помощью каких инструментов они были получены. Некоторые средства анализа входят в состав поставки инструмента.

2. Руководство пользователя

2.1. Запуск инструмента

Утилита командной строки `buildography` имеет интерфейс, аналогичный классическим отладчикам и трассировщикам. Она принимает в качестве аргументов программу и аргументы, с которыми её необходимо выполнить.

Программа может быть задана абсолютным или относительным путём до исполняемого файла, либо только именем. В последнем случае исполняемый файл ищется в директориях, перечисленных в значении переменной окружения `PATH` (значение по умолчанию см. в документации функции `execvp`).

```
buildography [options...] <program> [args...]
```

Утилита принимает следующие опции:

- `-d | --dump-dir <DIR>` — директория, куда будет помещён выходной файл. По умолчанию используется текущая директория.
- `-T | --threads <NUM>` — ограничить максимальное число потоков (по умолчанию стоит ограничение в 8 потоков).
- `-o | --output <NAME>` — явно указать выходной файл (по умолчанию см. раздел 2.2.2).
- `-e | --stderr <FILE>` — перенаправить диагностику трассировщика в файл (по умолчанию диагностика выводится в `stderr`).

Флаги `-d` и `-o` не могут быть использованы вместе.

Код возврата утилиты задаётся следующим образом:

1. Если произошла ошибка трассировки, код возврата устанавливается в 1;
2. Иначе, если трассируемая команда завершилась по сигналу, чей код равен `s`, код возврата устанавливается в `128 + s`;

3. В противном случае код возврата утилиты равен коду возврата команды.

В случае ошибки трассировки трассируемым процессам посылается сигнал SIGKILL.

2.1.1. Пример

Открытое ПО Postgres можно собрать, выполнив два шага: конфигурацию с помощью скрипта `configure` и сборки с помощью утилиты `make`.

Для перехвата всего процесса сборки и получения результатов в едином JSON файле нужно каким-либо способом объединить все команды сборки. Например, с помощью shell-скрипта.

```
git clone https://github.com/postgres/postgres.git
cd postgres

cat <<EOF > build.sh
#!/bin/bash -e
mkdir -p build
cd build
../configure --without-readline
make -j16
EOF
chmod +x build.sh

/opt/buildography/bin/buildography ./build.sh
```

Другой подход состоит в раздельном анализе шагов сборки. Поскольку в данном примере два шага, запустим инструмент дважды:

```
mkdir -p build2
cd build2
/opt/buildography/bin/buildography ../configure --without-readline
/opt/buildography/bin/buildography make -j16
```

В результате получим два JSON файла.

2.2. Формат выходного файла

Результат работы инструмента — единственный `.json` файл с информацией о процессах, порождённых в процессе сборки. Для удобства помимо сырой информации об индивидуальных процессах он содержит уже сгруппированные данные, полученные на её основе — поле `utilities`. Смысл полей раскрывается ниже.

Выходной JSON файл имеет следующую структуру:

```
{
    "buildography": { "major": "<ARG>", "minor": "<ARG>", "patch": "<ARG>",
"prerelease": "<ARG>" },
    "directory": "<PATH>",
    "commands": [ <ARGV...> ],
    "component_commands": [
        {
            "id": <ID>,
            "parent_id": <ID>,
            "starts_at": <TIMESTAMP>,
            "ends_at": <TIMESTAMP>,
            "command": [ <ARGV...> ],
            "dependencies": [ <FILESPEC...> ],
            "output": [ <FILESPEC...> ]
        },
```

```
        ...
    ],
    "utilities": [ <FILESPEC...> ]
}
```

Где

- `<FILESPEC> ::= {"seqno": <SEQNO>, "path": <PATH>, "hash": <HASH>, "is_text": <IS_TEXT>}`;
- `<SEQNO>` — уникальный идентификатор файла (число);
- `<PATH>` — абсолютный путь к файлу или директории (строка);
- `<HASH>` — хэш-сумма в шестнадцатеричном представлении (в данный момент используется 256-битный вариант ГОСТ Р 34.11-2012 — строка длины 64);
- `<IS_TEXT>` — целое число, равно 1, если файл является текстовым, иначе 0. Файл считается текстовым, если он не содержит байтов со следующими значениями: `0x00..0x06`, `0x0E..0x19`, `0x1C..0x1F`, `0x7F`;
- `<ARGV>` — аргумент командной строки (строка);
- `<ID>` — уникальный идентификатор команды (число);
- `<TIMESTAMP>` — Unix-время в микросекундах (число).

2.2.1. Значение полей выходного файла

- `buildography` — информация о версии инструмента. Содержит поля `major`, `minor`, `patch` (строки с числами), а также поле `prerelease` в пре-релизных сборках (в релизных отсутствует). Значение `prerelease` — идентификатор пре-релиза по SemVer: идентификаторы из ASCII-букв, цифр и дефиса, разделённые точками, например `pre` или `rc.1`.
- `directory` — рабочая директория, из которой была запущена исходная команда.
- `commands` — массив аргументов исходной команды (включая исполняемый файл — `argv[0]`).
- `component_commands` — массив объектов, описывающих вызовы программ, выполненные в процессе выполнения исходной команды. Вложенные вызовы также содержатся в этом массиве, но сам массив — «плоский». Каждый объект содержит следующие поля:
  - `id` — уникальный идентификатор образа процесса команды (не является `pid` процесса);
  - `parent_id` — идентификатор родительского образа процесса команды, отсутствует для корневых команд;
  - `starts_at` — время начала работы команды;
  - `ends_at` — время окончания работы команды;
  - `command` — массив аргументов командной строки (аналогично `commands`);
  - `dependencies` — массив объектов, каждый из которых представляет собой входной файл для данного запуска программы и имеет 4 поля:
    - `seqno` — уникальный идентификатор файла;
    - `path` — путь до файла;
    - `hash` — хэш-сумма содержимого этого файла;
    - `is_text` — является ли объект текстовым.
  - `output` — массив того же формата, но для выходных файлов.
- `utilities` — массив объектов, описывающих все вызванные программы, в формате, аналогичном `dependencies`.

2.2.2. Контрольные суммы

Для идентификации файлов используются контрольные суммы (хэш-суммы) их содержимого. Они вычисляются с помощью хэш-функции, определённой ГОСТ 34.11-2018 (ГОСТ Р 34.11-2012).

Название выходного файла составлено из хэш-суммы его содержимого и `.json`.

2.3. Средства анализа

Выходной JSON файл можно анализировать как вручную, так и создавая инструменты для автоматического исследования свойств сборки. Несколько таких инструментов входят в состав поставки Buildography.

Условием для корректной работы средств анализа является совпадение основных версий и подверсий Buildography во входных JSON файлах с текущей основной версией и подверсией инструмента. При несовпадении версий вызывается аварийный останов скрипта.

2.3.1. Определения

Задачи некоторых из этих инструментов включают поиск «листовых» файлов — таких, которые используются в процессе сборки, но которые в свою очередь не были созданы трассируемыми процессами. Для их обозначения введём следующее определение.

Входными файлами сборки будем называть файлы, которые были открыты на чтение и не были открыты на запись в процессе трассировки.

2.3.2. Сторонние бинарные файлы — `bxy_find_unbuilt_binaries`

Скрипт `bxy_find_unbuilt_binaries` выводит в файл перечень использованных при сборке «бинарных» файлов, *которые не были созданы в процессе этой сборки.* Мотивация применения — убедиться в том, что ПО собирается целиком из исходных файлов или выявить предсобранные зависимости.

```
usage: bxy_find_unbuilt_binaries [-h] [-o OUTPUT] input
```

«Бинарными» считаются файлы, для которых в JSON значение поля `is_text` равно 0.

Пример запуска:

```
json=8b5177e2bd383896983965e793bee81c19a358068d0d84460442603edc715979.json
bxy_find_unbuilt_binaries "$json" -o blobs.txt
```

В результате для примера сборки postgres `blobs.txt` содержит (приводится сокращённый список):

```
/usr/bin/install
/usr/lib/bfd-plugins/libdep.so
/usr/lib/crti.o
/usr/lib/gcc/x86_64-pc-linux-gnu/14.2.1/libgcc.a
/usr/lib/gcc/x86_64-pc-linux-gnu/14.2.1/liblto_plugin.so
/usr/lib/ld-linux-x86-64.so.2
/usr/lib/libacl.so.1.1.2302
/usr/lib/libc_nonshared.a
/usr/lib/libcrypt.so.2.0.0
/usr/lib/libc.so.6
/usr/lib/libguile-3.0.so.1.7.0
/usr/lib/libicudata.so.75.1
/usr/lib/libLLVM.so.18.1
/usr/lib/liblzma.so.5.6.2
/usr/lib/libmpfr.so.6.2.1
/usr/lib/libm.so.6
/usr/lib/libstdc++.so.6.0.33
/usr/lib/libxml2.so.2.13.3
/usr/lib/libz.so.1.3.1
/usr/lib/libzstd.so.1.5.6
/usr/lib/LLVMgold.so
/usr/lib/perl5/5.38/core_perl/auto/Cwd/Cwd.so
/usr/lib/perl5/5.38/core_perl/CORE/libperl.so
/usr/lib/Scrt1.o
...
```

2.3.3. Внешние исходные файлы — `bxy_find_external_sources`

Скрипт `bxy_find_external_sources` выводит в файл перечень файлов, использованных в процессе сборки и при этом:

1. не находящихся в заданных каталогах `SRC_DIR` и их подкаталогах;
2. не созданных в результате выполнения команд сборки;
3. являющихся «исходниками».

```
usage: bxy_find_external_sources [-h] [-d SOURCE_DIR] [-o OUTPUT] input
```

Под «исходниками» понимаются файлы, для которых в JSON значение поля `is_text` равно 1.

Мотивация применения — убедиться, что все исходные коды в процессе сборки извлекаются из заданных каталогов или выявить внешние по отношению к ним файлы с исходным кодом.

Пример запуска:

```
json=8b5177e2bd383896983965e793bee81c19a358068d0d84460442603edc715979.json
bxy_find_external_sources "$json" \
    -o external-src.txt \
    -d /mnt/co/postgres \
    -d /usr/lib \
    -d /usr/include
```

Возможное содержимое `external-src.txt` после запуска скрипта:

```
/etc/nsswitch.conf
/etc/passwd
/proc/602303/maps
...
/proc/628707/maps
/proc/sys/kernel/ngroups_max
/usr/bin/egrep
...
/usr/share/bison/m4sugar/foreach.m4
...
/usr/share/bison/skeletons/c-skel.m4
/usr/share/bison/skeletons/yacc.c
/usr/share/locale/locale.alias
/usr/share/perl5/core_perl/Carp.pm
...
/usr/share/perl5/core_perl/XSLoader.pm
```

2.3.4. Ингредиенты файла — `bxy_trace_origins`

Скрипт `bxy_trace_origins` по выходному файлу строит множество *входных файлов сборки* (см. раздел 2.3.1), «составляющих» его. В качестве выходного файла могут быть переданы в том числе любые промежуточные файлы, полученные в ходе сборки.

```
usage: bxy_trace_origins [-h] (-a BUILD_ARTIFACT | -H HASH | -s SEQNO)
                         [-o OUTPUT_FILE] [--hasher HASHER]
                         [-f FILTER_SUBTREE] [--with-intermediate]
                         [--format {plain,pdf}]
                         json
```

Обязательные аргументы:

- Путь до исходного JSON файла.

- Также обязательно наличие ровно одного из трёх аргументов-способов задания выходного файла, описанных ниже.

Задать выходной файл можно с помощью следующих опций:

- С помощью `--build-artifact` задаётся путь до артефакта сборки в файловой системе. Требует опцию `--hasher`.
  - С помощью `--hasher` задаётся командная строка для запуска утилиты, печатающей хэш-сумму переданного ей файла-аргумента. Этот параметр нужен, чтобы вычислить контрольную сумму переданного файла. Далее для поиска в JSON записей об этом файле он идентифицируется контрольной суммой.
- С помощью `--hash` непосредственно задаётся хэш артефакта сборки.
- С помощью `--seqno` задаётся уникальный идентификатор seqno артефакта сборки.

В случае, если заданным параметрам соответствует более одного выходного файла, скрипт завершит работу, выведя сообщение о множестве найденных кандидатов, для каждого из которых будут выведены seqno и путь. После выбора файла его можно будет указать с помощью `--seqno`.

Опциональные аргументы:

- С помощью `--output-file` задаётся путь до файла, в который будет записана информация о зависимостях сборки артефакта.
- С помощью `--filter-subtree` задаётся директория, по которой будут отфильтрованы найденные файлы: файл будет выведен в том случае, если он находится в заданном каталоге или его подкаталоге.
- Если передана опция `--with-intermediate`, скрипт помимо найденных исходных файлов выводит ещё и промежуточные зависимости.
- С помощью `--format` задаётся формат отчёта о зависимостях сборки артефакта. Доступны следующие форматы: обычный текст, PDF.

Пример запуска:

```
json=8b5177e2bd383896983965e793bee81c19a358068d0d84460442603edc715979.json
bxy_trace_origins \
    --input "$json" \
    --hasher "gost12-256-hash" \
    --build-artifact build/src/pl/plpgsql/src/plpgsql.so \
    --output-file trace-origins.txt
```

Возможное содержимое `trace-origins.txt` после запуска скрипта:

```
748d255fc...  /usr/share/locale/locale.alias
550a10d61...  /mnt/co/postgres/src/include/utils/plancache.h
5bdd142c6...  /mnt/co/postgres/src/include/access/xact.h
238ea9ca4...  /usr/lib/libz.so.1.3.1
c7f66e87b...  /usr/include/bits/types/__locale_t.h
1fd07fd0f...  /mnt/co/postgres/src/include/lib/simplehash.h
2753c6ed1...  /usr/include/bits/select.h
4a8dda1f5...  /mnt/co/postgres/src/include/common/kwlookup.h
c707bf1c1...  /mnt/co/postgres/src/include/access/tupmacs.h
b1bb13531...  /usr/include/bits/stdint-least.h
cdda67733...  /mnt/co/postgres/src/include/storage/procnumber.h
8fa04cb22...  /usr/include/bits/wordsize.h
2793fa39d...  /mnt/co/postgres/src/include/pg_config_ext.h
73e23d526...  /mnt/co/postgres/src/include/storage/relfilelocator.h
bfab053fa...  /mnt/co/postgres/src/include/access/tupdesc.h
5c641f350...  /usr/include/unistd.h
...
```

2.3.5. Объединение JSON файлов — `bxy_merge_json`

Скрипт `bxy_merge_json` позволяет объединить несколько JSON файлов в формате выходного файла Buildography (см. раздел 2.2) в один файл. По умолчанию файлы объединяются в одну сущность, если их хэш-суммы совпадают.

```
usage: bxy_merge_json [-h] [-o OUTPUT] [--basename-heuristic] [-f] input [input ...]
```

Обязательными аргументами скрипта являются пути к JSON файлам.

Опциональные аргументы:

- С помощью `--output` задаётся путь до объединенного JSON файла.
- С помощью `--basename-heuristic` включается эвристика равенства файлов при равенстве их базовых имён и хэш-сумм одновременно.

Скрипт объединяет файлы следующим образом:

- Поля `commands` и `directory` выставляются в `[]` и `""` соответственно.
- Поля `component_commands` и `utilities` объединяются в последовательном порядке, причём файлы получают новые идентификаторы `<SEQNO>`.

2.3.6. Множество входных файлов — `bxy_get_input_files`

Скрипт `bxy_get_input_files` выводит множество *входных файлов сборки* (см. раздел 2.3.1) в формате `{ <FILESPEC...> }` (см. раздел 2.2).

```
usage: bxy_get_input_files [-h] [-f FILTER_SUBTREE] -i INPUT [-o OUTPUT]
```

Обязательным аргументом скрипта является путь до исходного JSON файла (опция `--input`).

Опциональные аргументы:

- С помощью `--filter-subtree` задаётся директория, по которой будут отфильтрованы найденные файлы: файл будет выведен в том случае, если он находится в заданном каталоге или его подкаталоге.
- С помощью `--output-file` задаётся путь до файла, в который будет записано множество входных файлов сборки.

Пример запуска:

```
json=8b5177e2bd383896983965e793bee81c19a358068d0d84460442603edc715979.json
bxy_get_input_files \
    --input "$json" \
    --filter-subtree /mnt/co/postgres/src/pl/plpgsql \
    --output-file input-files.json
```

Возможное содержимое `input-files.json` после запуска скрипта:

```
[
  {
    "seqno": 44,
    "path": "/mnt/co/postgres/src/pl/plpgsql/src/generate-plerrcodes.pl",
    "hash": "06f8c3360..."
  },
  {
    "seqno": 77,
    "path": "/mnt/co/postgres/src/pl/plpgsql/src/pl_reserved_kwlist.h",
    "hash": "447a7d2b1..."
  },
```

```
  {
    "seqno": 78,
    "path": "/mnt/co/postgres/src/pl/plpgsql/src/pl_unreserved_kwlist.h",
    "hash": "cb014ebeb..."
  },
  ...
]
```

2.3.7. Разница между JSON файлами — `bxy_diff`

Скрипт `bxy_diff` позволяет вычислить разницу между JSON файлами в формате выходного файла Buildography (см. раздел 2.2) с заданной точностью.

```
usage: bxy_diff [-h] [-o OUTPUT] [-r RENAME_BUILD_DIR]
                [-e EQCLASS_MASK] [-E EQCLASS_REGEX]
                [-s SKIP_MASK] [-S SKIP_REGEX]
                [--ignore-hash-diffs] [--ignore-rwfiles]
                [--remove-duplicates]
                [-c IGNORE_COMMANDS_LIKE] [-C ACCEPT_ONLY_COMMANDS_LIKE]
                json_1 json_2
```

Обязательными аргументами скрипта являются пути к JSON файлам.

Опциональные аргументы:

- С помощью `--output` (`-o`) задаётся путь до JSON файла, представляющего собой разницу во входных файлах. По умолчанию печатается в стандартный поток вывода.
- С помощью `--rename-build-dir` (`-r`) задаётся имя директории, на которое при сравнении требуется заменить имена сборочных директорий, указанных в поле `directory`. Опция полезна в случае, если сборки происходили в разных директориях, и разница в именах директорий не имеет значения. Например, если первая сборка происходила в директории `/x/y/a`, вторая — в `/x/y/b`, то, применив опцию `-r /x/y/c`, можно добиться того, что `/x/y/a/source.c` и `/x/y/b/source.c` будут считаться одним и тем же именем файла.
- С помощью опций `--eqclass-mask` (`-e`) и `--eqclass-regex` (`-E`) задаются классы эквивалентности в виде файловых масок и регулярных выражений соответственно. Файловой маской является последовательность символов, в которой * соответствует произвольному числу любых символов, ? - одному любому символу, остальные символы соответствуют сами себе. Компоненты команды (в частности имена файлов), соответствующие одному классу эквивалентности, при сравнении считаются одинаковыми. Если компонент команды соответствует нескольким классам, то он принимается относящимся только к тому из них, опция, соответствующая которому, была передана раньше. Если класс задан регулярным выражением, то учитывается полное соответствие строки регулярному выражению (подобно эффекту опции `-x` команды `grep`). Например, если классы заданы подстрокой `-e *x* -E .*y.*`, то компоненты `xa.c` и `xy.c` считаются одинаковыми, а `ay.c` и `xy.c` — разными.
- С помощью опций `--skip-mask` (`-s`) и `--skip-regex` (`-S`) задаются фильтры для компонентов команды, которые исключаются из неё при сравнении. Например, если задан фильтр `-s -O2`, то команды `gcc -O2 hello.c` и `gcc hello.c` будут считаться одинаковыми.
- Опция `--ignore-hash-diffs` позволяет игнорировать различия в хэшах файлов.
- Опция `--ignore-rwfiles` позволяет исключать из сравнения файлы, которые входят одновременно в `dependencies` и `output` одной команды.
- Опция `--remove-duplicates` позволяет удалять повторяющиеся команды (с точностью до классов эквивалентности компонентов и фильтров исключений).
- С помощью опции `-c` можно исключать из сравнения команды, начинающиеся с определённого префикса. Например, с помощью `-c bash,-c` можно исключить из сравнения все команды вида `bash -c <args>`.

- С помощью опции `-C` можно включать в сравнение только команды, начинающиеся с определённого префикса. Например, с помощью `-C gcc` можно построить разницу только в командах вида `gcc <args>`.

Далее в этом разделе будем называть массивы компонентов команд и словари файл-хэш эквивалентными, если они, после замены всех подстрок, равных имени сборочной директории, на переданное с помощью `-r` имя и после исключения всех элементов, соответствующих фильтрам, переданным через `-s` и `-S`, совпадают с точностью до классов эквивалентности (введённых с помощью `-e` и `-E`).

На выходе скрипт строит JSON файл, повторяющий структуру выходного файла Buildography (см. раздел 2.2) со следующими отличиями и особенностями:

- Исключается поле `buildography`.
- Если сборочные директории совпадают (до переименования), создаётся поле `= directory` с соответствующим значением. В противном случае создаются поля `< directory` и `> directory`, содержащие имена сборочных директорий первой и второй в порядке указанных JSON файлов сборок соответственно.
- `component_commands` представляет собой массив объектов, представляющих команды. Каждая команда имеет:
  - Ровно одно из трёх полей: `= command`, `< command`, `> command`. Существование первого поля указывает на то, что данный объект является результатом сравнения двух команд, массивы `command` которых эквивалентны, а словари `dependencies` и/или `output` неэквивалентны. Тогда значением поля является обобщённое представление команды, в которое подставлено новое значение сборочной директории вместо старых, удалены компоненты, соответствующие фильтрам исключений, а компоненты заменены на представления классов эквивалентности. Например для команд `gcc /home/build2/main.c -S -o /tmp/1111.asm` и `gcc /home/build1/main.c -S -o /tmp/2222.asm` при соответствующих аргументах может быть сгенерировано представление `gcc /home/build/main.c -S -o /tmp/*.asm`. Если объект имеет второе поле, то это уникальная команда из первой сборки, не имеющая эквивалентной из второй сборки. Её представление не обобщается и переписывается как есть, но с исключением полей seqno. Аналогично для третьего поля.
  - Массив `dependencies` с объектами с полями `path` и `hash`. Если объект соответствует двум командам с эквивалентными `command`, то поле `path` содержит строки вида `< filename` и `> filename`, соответствующие зависимостям только из первой или второй команды соответственно. Если объект соответствует уникальной команде из одной из сборок, то объект повторяет исходный, но с исключением полей seqno.
  - Массив `output` аналогично `dependencies`.
- Массив `utilities` содержит объекты как в `dependencies`. Поле `path` этих объектов содержит строки вида `< filename` и `> filename`, соответствующие утилитам только из первой или второй сборки соответственно.

Если классы эквивалентности и исключаемые компоненты не известны заранее, скрипт предполагается использовать интерактивно. Например, в качестве аргументов первого запуска скрипта достаточно передать пути до сравниваемых файлов, постепенно дополняя строку запуска, например, классами эквивалентности, исключаемыми компонентами и исключаемыми командами.

Пример запуска: пусть имеется две компиляции двух разных тривиальных программ на языке Си: `gcc main.c` в одной директории. При запуске без дополнительных параметров можно получить следующий вывод:

```
$ bxy_diff 1.json 2.json
...
{
  "< command": [
    "as",
    "-o",
    "/tmp/cctS8jOI.o",
```

```
    "/tmp/ccwXAX1F.s"
  ],
  ...
}
...
{
  "> command": [
    "as",
    "-o",
    "/tmp/cclJk2op.o",
    "/tmp/cchx07S6.s"
  ],
  ...
}
...
```

Здесь видно две уникальных команды, которые отличаются именами временных исходных файлов на языке ассемблера и объектных файлов. Можно определить два класса эквивалентности: первый для временных объектных файлов, второй для временных файлов на языке ассемблера, добавив их в аргументы скрипта:

```
$ bxy_diff 1.json 2.json -e '/tmp/*.o' -e '/tmp/*.s'
...
{
  "= command": [
    "as",
    "-o",
    "/tmp/*.o",
    "/tmp/*.s"
  ],
  "dependencies": [
    {
      "path": "> /tmp/cchx07S6.s",
      "hash": "dd9891da5..."
    },
    {
      "path": "< /tmp/ccwXAX1F.s",
      "hash": "3080fea88..."
    }
  ],
  "output": [
    {
      "path": "> /tmp/cclJk2op.o",
      "hash": "5dc133276..."
    },
    {
      "path": "< /tmp/cctS8jOI.o",
      "hash": "7ea68de3c..."
    }
  ]
},
...
```

Теперь скрипт рассматривает команды как эквивалентные по командной строке, однако программы являются семантически разными, вследствие чего разница в `dependencies` и `output` непустая — отличаются хэши файлов.

Отличие хэшей файлов можно игнорировать, передав в строке опцию `--ignore-hash-diffs`:

```
$ bxy_diff 1.json 2.json -e '/tmp/*.o' -e '/tmp/*.s' --ignore-hash-diffs
```

Теперь записей, соответствующих вызову ассемблера, в выводе не будет, поскольку команды считаются полностью эквивалентными.

2.3.8. Классификация открытых файлов — `bxy_classify_files`

Скрипт классифицирует открытые файлы по категориям и выводит статистику — в зависимости от опций полный список файлов для каждой категории или число файлов в каждой категории. При этом файлы, хэши которых совпадают, отождествляются и попадают в одни и те же категории, что позволяет правильно учесть файлы, полученные в результате копирования.

```
usage: bxy_classify_files [-h] [--categories CATEGORIES] [--stat]
                          [--allow-multiple-categories] [--locale LOCALE]
                          json
```

Обязательным аргументом является путь к входному JSON файлу.

Опциональные аргументы:

- С помощью `--categories` задаётся путь к YAML файлу, задающему правила, по которым описываются категории файлов. Если опция не задана, используется файл по умолчанию.
- С помощью `--stat` для каждой категории выводится только число относящихся к категории файлов. Если опция не задана, выводится полный список принадлежащих ей файлов.
- Если не указана `--allow-multiple-categories`, каждый файл сопоставляется только одной подходящей категории, которая стоит первой по списку в файле категорий, иначе каждой подходящей.
- С помощью `--locale` задаётся идентификатор региональных настроек для вывода описаний категорий (например, `ru_RU` или `en_US`). По умолчанию региональные настройки определяются автоматически.

Категории задаются в файле формата YAML. По умолчанию используется лежащий в папке `buildography/share` файл `file-categories.yaml`, он может быть переопределён с помощью опции `--categories`.

Файл категорий должен содержать массив `categories` объектов из четырёх полей:

- `description` — описание категории, используется для вывода статистики. Может быть указано в виде строки или словаря, ключами которого являются допустимые идентификаторы региональных настроек (например, `ru_RU` или `en_US`) или их префиксы (`ru`, `en`). Если выбранные региональные настройки не соответствуют ни одному из описаний, выбирается английский вариант описания;
- `pattern` — регулярное выражение — фильтр файлов по их пути;
- `utility` — регулярное выражение — фильтр файлов по пути к утилите, которая их открывала;
- `is_output` — логическое значение, если `true`, то файл является выходным файлом утилиты, иначе входным, это необязательное поле, значение по умолчанию — `false`.
- `is_text` — необязательное поле с логическим значением, если `true`, то файл считается текстовым, если `false` — бинарным, если поле не указано, категория допускает оба варианта.

В качестве примера использования рассмотрим анализ процесса компиляции одного исходного файла на языке Си компилятором GCC. Пусть данные о сборке расположены в файле `1.json`. Составим свой файл с описанием классификации файлов:

```
categories:
  - pattern: '.*[.]c'
    utility: '(.*/)?(cc|cc1)(-\d+([.]\d+)*)?'
    description: 'Исходный код C'
    is_text: true
```

Сейчас он состоит только из одной категории — файлы на языке Си. Запустим анализатор, указав наш файл классификации:

```
$ bxy_classify_files.py 1.json --categories cat.yaml
=== Исходный код C ===
  - /mnt/co/main.c (seqno: 14)
=== Неопознанные файлы ===
...
  - /tmp/ccthzwi7.s (seqno: 5)
  - /tmp/ccthzwi7.s (seqno: 15)
...
```

Имена файлов в выводе могут встречаться несколько раз, поскольку в процессе сборки содержимое файлов может изменяться, и файл с новым содержимым мы уже считаем другим файлом.

Вывод категории `Uncategorized files` оказался довольно объёмным, чтобы увидеть только число файлов в каждой категории, запустим скрипт, указав опцию `--stat`:

```
$ bxy_classify_files.py 1.json --categories cat.yaml --stat
Исходный код C: 1
Неопознанные файлы: 40
```

Здесь мы видим, что используя данный файл классификации, анализатор не смог сопоставить какой-либо категории 40 файлов.

Увидев список файлов, можно улучшить классификацию, добавив в неё новый класс, в нашем случае им будет, например, класс файлов на языке ассемблера:

```
  - pattern: '.*[.]s'
    utility: '(.*/)?(as)(-\d+([.]\d+)*)?'
    description: 'Ассемблерный код'
    is_text: true
```

Повторный запуск команды позволит увидеть статистику о новой категории:

```
$ bxy_classify_files.py 1.json --categories cat.yaml --stat
Исходный код C: 1
Ассемблерный код: 1
Неопознанные файлы: 39
```

В новую категорию вошло только одно содержимое файла `/tmp/ccthzwi7.s`.

В комплекте со скриптом поставляется также файл с описанием категорий по умолчанию, его можно использовать, убрав аргумент `--categories`:

```
$ bxy_classify_files.py 1.json
Исходный код C: 1
Заголовочный файл C/C++: 1
Ассемблерный код: 1
Статическая библиотека: 2
Разделяемая библиотека: 10
Объектный код: 1
Неопознанные входные двоичные файлы: 16
Неопознанные выходные двоичные файлы: 4
Неопознанные входные текстовые файлы: 3
Неопознанные выходные текстовые файлы: 2
```

2.3.9. Обнаружение непосредственных входных файлов компиляторов — `bxy_get_compilable_sources`

Скрипт выводит список исходных файлов, поступающих на вход компиляторам. В список не входят временные файлы, создаваемые в процессе компиляции, а также файлы, из которых получаются исходные путём внешнего препроцессирования, например, подстановкой значений определённых констант. Скрипт опирается на априорное знание о типах утилит, запущенных в процессе сборки, описанное в YAML файле. Также скрипт выводит список неклассифицированных утилит, которые возможно являются компиляторами или компоновщиками. Скрипт считает *подозрительной* любую неклассифицированную утилиту, которая читает ранее созданный в рамках сборки бинарный файл или производит бинарный файл на выходе.

```
usage: bxy_get_compilable_sources [-h] [-u UTILITIES] [-f FILTER_SUBTREE]
                                  [-o OUTPUT_FILE]
                                  json
```

Обязательным аргументом является путь к входному JSON файлу.

Опциональные аргументы:

- С помощью `--utilities` задаётся путь к YAML файлу с классификацией утилит. Если опция не задана, используется файл по умолчанию.
- С помощью `--filter-subtree` задаётся директория, по которой будут отфильтрованы найденные файлы: файл будет выведен в том случае, если он находится в заданном каталоге или его подкаталоге.
- С помощью `--output-file` задаётся путь до файла, в который будет записана полученная информация.

Классификация утилит задаётся в файле формата YAML. По умолчанию используется лежащий в папке `buildography/share` файл `utilities.yaml`, он может быть переопределён с помощью опции `--utilities`.

Файл категорий должен содержать объект `utilities` из пяти необязательных полей:

- `preprocessors` — список (массив) препроцессоров;
- `compilers` — список компиляторов;
- `linkers` — список компоновщиков;
- `interpreters` — список интерпретаторов;
- `other` — список утилит, не входящих в вышеперечисленные классы.

Каждая утилита задаётся командой — именем бинарного файла (basename).

В качестве примера использования рассмотрим сборку тривиального проекта:

```
.PHONY: all
all:
        /x/y/insert main.c.in main.c
        gcc -E main.c -o main.i
        gcc main.i -o prg
```

Здесь `/x/y/insert` — некоторая утилита, которая подставляет определённые данные в шаблон исходного файла.

Получим `build.json` и отдадим его скрипту анализа. В качестве классификатора утилит укажем `utilities.yaml` с пустым содержанием:

```
utilities: {}
```

Запустим скрипт:

```
$ bxy_get_compilable_sources build.json -u utilities.yaml
```

Поскольку мы не определили ни одного компилятора, список найденных файлов будет пустым. Однако скрипт найдёт потенциальные компиляторы и компоновщики. Рассмотрим вывод команды:

```
=== Sources ===

=== Utilities to define ===
- ld <- collect2 <- gcc <- make
- as <- gcc <- make
```

Скрипт выделил `ld` и `as` как *подозрительные* утилиты. Однако, поскольку они могут быть лишь частью дерева процессов компилятора, вместе с ними выводится до трёх предков. Можно заметить, что *подозрительные* утилиты являются частью `gcc`, который мы «забыли» обозначить как компилятор:

```
utilities:
  compilers:
    - gcc
```

Следующий запуск покажет искомые исходные файлы:

```
$ bxy_get_compilable_sources build.json -u utilities.yaml

=== Sources ===
- /build/dir/main.c (seqno 12)
- /etc/locale.alias (seqno 14)
- /usr/include/stdc-predef.h (seqno 24)
- /usr/lib/gcc/x86_64-linux-gnu/13/libgcc_s.so (seqno 48)
- /usr/lib/x86_64-linux-gnu/libc.so (seqno 50)
```

`main.c.in` не был выведен, поскольку не является входом компилятора. `main.i` является результатом препроцессирования (скрипт сделал такой вывод, поскольку `main.i` является выходным файлом компилятора).

Часть из них являются системными. Чтобы очистить вывод от системных файлов, воспользуемся опцией `-f`:

```
$ bxy_get_compilable_sources build.json -u utilities.yaml -f /build/dir

=== Sources ===
- /build/dir/main.c (seqno 12)
```

2.3.10. Распечатка дерева команд сборки — `bxy_print_tree`

Скрипт выводит дерево команд сборки в текстовом виде, похожем на вывод команды `pstree`.

```
usage: bxy_print_tree [-h] [-a] json
```

Обязательным аргументом является путь к входному JSON файлу.

Опциональные аргументы:

- С помощью `--ascii` символы, отображающие дерево, являются ASCII символами. По умолчанию используется Unicode.

Пример вывода дерева команд тривиальной компиляции с помощью gcc:

```
$ bxy_print_tree gcc.json
[0]gcc─┬─[1]cc1
       ├─[2]as
       └─[3]collect2──[4]ld
```

ASCII-вариант:

```
$ bxy_print_tree gcc.json -a
[0]gcc-+-[1]cc1
       |-[2]as
       `-[3]collect2---[4]ld
```

2.3.11. Зависимости по данным между выбранными командами — `bxy_dependency_dag`

Скрипт строит *сокращённый ациклический ориентированный граф (DAG) зависимостей по данным.*

Вершинами этого графа являются *приведённые выбранные команды*, которые получаются в результате процедуры **свёртки** исходного JSON, которая заключается в следующем:

1. Выбираются команды, которые удовлетворяют некоторому фильтру.
2. Если команда A имеет подкоманды, то все транзитивно порождённые ей подкоманды удаляются, их `dependencies` и `output` добавляются в соответствующие поля A, а командные строки сохраняются в новом поле A. Если у выбранной команды есть выбранный потомок, то он таким же способом удаляется в пользу предка.
3. Оставшиеся выбранные команды становятся приведёнными выбранными командами.

Свёртка удобна, например, для случаев, когда выбранной командой является драйвер компилятора (например, `gcc` или `clang++`), чтобы абстрагироваться от деталей процесса компиляции и строить зависимости между вызовами драйверов.

Дуга между двумя приведёнными выбранными командами A и B ($A \neq B$) проводится тогда и только тогда, когда существует последовательность команд $C_1, C_2, ...C_k$, такая что $C_1 = A$, $C_k = B$, $C_2 ... C_{k-1}$ не являются выбранными, и для каждой пары $< C_i, C_{i+1} >$ существует файл, который является выходным (`output`) для C_i и входным (`dependencies`) для C_{i+1}.

Выбранные команды задаются списком регулярных выражений для имён (basename) утилит. Предполагаемый сценарий применения — определение зависимостей по данным между командами компиляции и компоновки.

```
usage: bxy_dependency_dag [-h] -s SELECTED_COMMANDS
                          [--disable-factorization-warnings] [-o OUTPUT]
                          json
```

Обязательные аргументы:

- Путь до входного JSON-файла.
- С помощью `--selected-commands` задаётся путь к YAML-файлу с фильтром команд. Файл с фильтром команд должен содержать массив `selected_commands` регулярных выражений. В качестве примера файла с фильтром выражений в поставку включён файл `c-cxx-build-commands.yaml`, расположенный в папке `buildography/share`.

Опциональные аргументы:

- С помощью `--disable-factorization-warnings` отключается вывод предупреждений о командах, которые, возможно, имеет смысл сделать выбранными, чтобы уменьшить размер выходного графа (факторизовать вывод).
- С помощью `--output-file` задаётся путь до файла, в который будет записана полученная информация. По умолчанию используется стандартный поток вывода.

Выходной файл имеет формат JSON и содержит описание построенного DAG. Схема выходного файла:

```
[
    {
        "id": <ID>,
        "command": [ <ARGV...> ],
        "subcommands": [ [ <ARGV...> ], ... ],
        "producer_ids": [ <ID...> ],
        "dependencies": [ <FILESPEC...> ],
        "output": [ <FILESPEC...> ]
    },
    ...
]
```

Где

- `ID` — уникальный идентификатор команды (число);
- `ARGV` — аргумент командной строки (строка);
- `FILESPEC` — формат описания файла, совпадает с форматом в выходном JSON трассировщика (см. раздел 2.2).

Значения полей:

- `id` — идентификатор команды, совпадает с `id` соответствующей команды из входного JSON;
- `command` — массив аргументов исходной команды (включая исполняемый файл — `argv[0]`), совпадает с `command` соответствующей команды из входного JSON;
- `subcommands` — массив командных строк подкоманд. Для команд-посредников всегда пуст;
- `producer_ids` — список идентификаторов (`id`) команд, от которых зависит данная команда;
- `dependencies` — массив `dependencies` команды из исходного JSON, дополненный в результате свёртки;
- `output` — массив `output` команды из исходного JSON, дополненный в результате свёртки.

Узлы DAG в массиве для удобства анализа расположены в топологическом порядке.

В качестве примера рассмотрим сборку, заданную следующим `Makefile`:

```
.PHONY: all

all:
  cp ret0.c main.c
  cp foo.c bar.c
  gcc -c main.c -o main.o
  gcc -c bar.c -o bar.o
  ar rcs main.a main.o bar.o
  gcc main.a -o main.exe
  cp main.exe ret0.exe
```

В качестве выбранной утилиты зададим `gcc` и `ar`:

```
selected_commands:
  - ar
  - gcc
```

Получив с помощью трассировщика `build.json`, передадим его на вход скрипту:

```
$ bxy_dependency_dag build.json -s commands.yml -o output.json
```

Результат будет отражён в файле `output.json`:

```
[
  {
    "id": 3,
    "command": ["gcc", "-c", "main.c", "-o", "main.o"],
    "subcommands": [
      ["/usr/libexec/gcc/x86_64-linux-gnu/13/cc1", ...],
      ["as", ...]
    ],
    "producer_ids": [],
    "dependencies": [{ "path": "/build/main.c", ... }, ...],
    "output": [{ "path": "/build/main.o", ... }, ...]
  },
  {
    "id": 6,
    "command": ["gcc", "-c", "bar.c", "-o", "bar.o"],
    "subcommands": [
      ["/usr/libexec/gcc/x86_64-linux-gnu/13/cc1", ...],
      ["as", ...]
    ],
    "producer_ids": [],
    "dependencies": [{ "path": "/build/bar.c", ... }, ...],
    "output": [{ "path": "/build/bar.o", ... }, ...]
  },
  {
    "id": 9,
    "command": ["ar", "rcs", "main.a", "main.o", "bar.o"],
    "subcommands": [],
    "producer_ids": [3, 6],
    "dependencies": [
      { "path": "/build/bar.o", ... },
      { "path": "/build/main.a", ... },
      { "path": "/build/main.o", ...},
      ...
    ],
    "output": [{ "path": "/build/main.a", ... }, ...]
  },
  {
    "id": 10,
    "command": ["gcc", "main.a", "-o", "main.exe"],
    "subcommands": [
      ["/usr/libexec/gcc/x86_64-linux-gnu/13/collect2", ...],
      ["/usr/bin/ld", ...]
    ],
    "producer_ids": [9],
    "dependencies": [{ "path": "/build/main.a", ... }, ...],
    "output": [{ "path": "/build/main.exe", ...}, ...]
  }
]
```

Рассмотрим узел с `id = 3`. Массив `subcommands` отражает тот факт, что в поддереве были вызваны утилиты `cc1` и `as`. Поддерево команд читало файл `/build/main.c` и записывало в файл `/build/main.o`. Этот узел не читает выходные файлы других узлов графа. Узел с `id = 6` построен аналогично.

Массив `subcommands` пуст — `ar` не вызывал какие-либо подкоманды. Содержимое массива `producer_ids` означает, что эта команда зависит по данным от двух команд компиляции.

Запись о команде с `"id": 10` в `subcommands` содержит вызовы `collect2` и `ld` и указывает на зависимость от команды архивации.

Если убрать `ar` из списка выбранных команд, то запись для `ar` исчезнет из выходного JSON, а массив `producer_ids` записи с `"id": 10` будет содержать значения `[3, 6]`, поскольку существуют цепочки запись-чтение $3 \to 9 \to 10$ и $6 \to 9 \to 10$, в которых все вершины (9), кроме начальных и конечных, не выбраны.

2.3.12. Выделение флагов компиляции и компоновки — `bxy_parse_flags`

Скрипт выделяет использованные наборы флагов компиляции и компоновки, а также определяет между ними следующую взаимосвязь: набор флагов компиляции считается связанным с набором флагов компоновки, если в процессе сборки хотя бы один объектный файл был собран с данным набором флагов компиляции и участвовал в компоновке с данным набором флагов компоновки.

Входной формат скрипта совпадает с выходным форматом `bxy_dependency_dag` (см. раздел 2.3.11), выходной формат — с входным форматом `test_build` (см. раздел 2.4.1).

```
usage: bxy_parse_flags [-h] -r RULES [--deduplication-mode DEDUPLICATION_MODE]
                       [-o OUTPUT]
                       json
```

Обязательные аргументы:

- Путь до входного JSON-файла в выходном формате `bxy_dependency_dag` (см. раздел 2.3.11).
- С помощью `--rules` задаётся путь к YAML-файлу с правилами разбора командных строк. В качестве примера файла с правилами в поставку включён файл `c-cxx-rules.yaml`, расположенный в папке `buildography/share`.

Опциональные аргументы:

- С помощью `--deduplication-mode` задаётся один из способов дедупликации команд: `full` (полная), `partial` (частичная), `none` (не производится). По умолчанию выполняется полная дедупликация.
- С помощью `--output-file` задаётся путь до файла, в который будет записана полученная информация. По умолчанию используется стандартный поток вывода.

Верхнеуровневый объект файла с правилами разбора командных строк представляет собой словарь, ключами которого являются классы команд, а значениями — правила их анализа.

Пара ключ-значение имеет следующий формат:

```yaml
<CMD_CLASS>:
  basename: <BASENAME_REGEX>
  options: # optional
    <OPTION>: <OPTION_TYPE>
    ...
  requires: # optional
    - options: [<OPTION>...]    # optional
      no_options: [<OPTION>...] # optional
    ...
  type: <TYPENAME> # optional
  types:           # optional, conflicting with type
    <TYPENAME>:
      - components: [<CMD_CLASS>...] # optional
        options: [<OPTION>...]       # optional
        no_options: [<OPTION>...]    # optional
      ...
    ...
```

Где:

- `<CMD_CLASS>` — класс команды (строка).
- `BASENAME_REGEX` — регулярное выражение (строка).

- `OPTION` — опция командной строки (строка).
- `OPTION_TYPE` — тип опции (строка или пустое значение). Может принимать одно из следующих значений:
  1. `noarg` — опция без аргументов (например `-opt`).
  2. `joined` — единственный аргумент опции сконкатенирован с опцией (например `-opt<arg>`).
  3. `separate` — единственный аргумент опции идёт сразу же за ней отдельным элементом командной строки (например `-opt <arg>`).
  4. `joined_or_separate` — единственный аргумент опции может быть как сконкатенирован с ней, так и идти сразу же за ней отдельным элементом.
  5. пустое значение — используется значение по умолчанию (`noarg`). Пример использования можно увидеть в примерах ниже.
- `TYPENAME` — тип команды (строка). Может иметь одно из следующих значений:
  1. `preprocessor` — данная команда считается вызовом препроцессора.
  2. `compile_asm` — данная команда считается компиляцией, результат компиляции — текстовый файл на языке ассемблера.
  3. `compile_obj` — данная команда считается компиляцией, результат компиляции — объектный файл.
  4. `compile_exe` — данная команда считается компиляцией, результат компиляции — исполняемый файл.
  5. `assemble` — данная команда считается ассемблированием.
  6. `link` — данная команда считается компоновкой.
  7. `archive` — данная команда считается архивом или частичной компоновкой.
  8. `skip` — данную команду следует пропустить при анализе.

Поля значения имеют следующее назначение:

- `basename` представляет собой регулярное выражение, которому сопоставляется имя файла утилиты команды.
- `options` — необязательное поле, которое содержит перечень (словарь) опций утилиты. В качестве ключа указывается имя опции, в качестве значения — её тип.
- `requires` — необязательное поле, которое содержит массив правил. Чтобы команду можно было отнести к данному классу `<CMD_CLASS>`, требуется, чтобы она удовлетворяла хотя бы одному из правил. Правило представляет собой объект с двумя полями: `options` — список опций, определённых в `options`, каждая из которых должна быть обнаружена в командной строке, и `no_options` — список опций, определённых в `options`, ни одна из которых не должна быть обнаружена в командной строке. Каждое из полей является необязательным, если представлены оба, то должны выполняться одновременно условия для каждого из полей.
- `type` — необязательное поле, которое, при наличии, автоматически определяет тип команды как `<TYPENAME>` в случае соответствия команды `<CMD_CLASS>`.
- `types` — необязательное поле, которое, при наличии, задаёт правила определения типа утилиты. Правила представлены объектом, ключами которого являются допустимые типы команды `<TYPENAME>`, а значениями — массивы правил, как для `requires`, с добавлением ещё одного необязательного поля — `components`, которое задаёт множество классов подкоманд, которое должно совпадать с вычисленным множеством классов подкоманд (компонент) основной команды.

`type` и `types` для одного класса не могут быть определены одновременно. При этом допускается отсутствие обоих полей, тогда такой класс команд не предполагает назначение типа команды. Тип назначается только основным командам. Таким образом, подобным классам команд могут соответствовать только подкоманды.

Кроме того, допустим сокращённый вариант:

```
<CMD_CLASS>: <BASENAME_REGEX>
```

Он является сокращением для следующей записи в полном варианте:

```
<CMD_CLASS>:
  basename: <BASENAME_REGEX>
```

Скрипт обрабатывает команды в порядке их следования во входном файле. Сначала он команде и каждой подкоманде сопоставляет класс. Для этого имя файла утилиты должно соответствовать `<BASENAME_REGEX>`, а сама команда при наличии поля `requires` — одному из перечисленных в нём правил. Подкоманда должна соответствовать не более чем одному классу, а основная команда — ровно одному. Для каждой основной команды определяется множество компонент, которое представляет собой множество классов его подкоманд.

После этого определяется тип основной команды. Команда должна соответствовать ровно одному типу.

На основе вычисленных типов скрипт строит множества команд компиляции и компоновки и взаимосвязи с ними. При этом из выходных команд удаляются все опции, перечисленные в поле `options`, их аргументы при наличии, а также позиционные аргументы, которые вычисляются путём поиска совпадений в командных строках и именах открытых командой файлов.

Списки команд дедуплицируются, если дедупликация не выключена пользователем. В полном режиме совпадающие команды одного выходного типа (компиляция или линковка) склеиваются, причём результат склейки наследует все связи исходных команд с командами другого типа с поправкой на их склейку. В частичном режиме склеиваются только команды компиляции, причём только те, у которых полностью совпадает список связанных с ними компоновок. Частичный режим позволяет сохранить информацию о структуре сборки.

Выходной файл представляет собой последовательность строк, каждая из которых описывает одну команду компиляции или компоновки и удовлетворяет формату JSON. Схема строки:

```
{"type": <OUT_TYPE>, ("id": <OUT_ID> | "input": [<OUT_ID>...]), "executable":
<EXE_NAME>, "arguments": [<ARGV>...]}
```

Где

- `OUT_TYPE` — тип команды (строка): `compile` либо `link`;
- `OUT_ID` — уникальный идентификатор команды (строка);
- `EXE_NAME` — путь или имя исполняемого файла (строка);
- `ARGV` — аргумент командной строки (строка).

Значения полей:

- `type` — тип команды, компиляция или компоновка;
- `id` — уникальный идентификатор команды, определён только для компиляций;
- `input` — массив связанных команд компиляции, только для компоновок;
- `executable` — команда вызова компилятора или компоновщика;
- `arguments` — массив аргументов компилятора или компоновщика.

Для примера зададим правила следующим образом:

```
ar:
  basename: ar
  type: archive
as: as
cc1: cc1(plus)?
gcc:
  basename: (([^-]+-){3,4})?g(cc|[+][+])(-[0-9]+)?
  options:
    -c:
    -o: joined_or_separate
```

```
  types:
    compile_obj:
      - components: [cc1, as]
        options: [-c]
    link:
      - components: [ld]
ld: ld
```

Здесь определено пять классов утилит:

- `ar` — архивы. Всегда имеют тип `archive`.
- `as` — ассемблер. Им не присваивается тип, поскольку в рамках примера мы ожидаем, что он будет встречаться только в подкомандах.
- `cc1` — вызовы фронтенда `gcc`. Аналогично не получают тип.
- `gcc` — вызовы драйвера `gcc`. В рамках примера будем различать только компиляцию в объектные файлы (`compile_obj`), которую мы определяем по наличию подкоманд классов `as` и `cc1` и отсутствию подкоманд других классов, а также наличию опции `-c`, и компоновку (`link`), которую мы определяем по наличию подкоманды класса `ld` и отсутствию подкоманд других классов.
- `ld` — вызов компоновщика. Не получает тип, только для подкоманд.

Для класса `gcc` также указана опция `-o`, которая может иметь аргумент через пробел или слитно с опцией: она не участвует ни в одном из правил, но приведена здесь, для того чтобы быть удалённой.

Для обоих примеров, приведённых для `bxy_dependency_dag` (см. раздел 2.3.11), скрипт может сгенерировать следующие командные строки:

```
{"type": "compile", "id": "3", "executable": "gcc", "arguments": []}
{"type": "link", "input": ["3"], "executable": "gcc", "arguments": []}
```

Все аргументы команд были убраны, поскольку они входят в `options` класса `gcc`. Команда с `id: "6"` удалена в пользу `id: "3"`, поскольку они совпадают с точностью до удалённых аргументов.

Можно добавить другие опции в командные строки `gcc`, например так:

```
.PHONY: all

all:
  cp ret0.c main.c
  cp foo.c bar.c
  gcc -O1 -c main.c -o main.o
  gcc -O1 -c bar.c -o bar.o
  ar rcs main.a main.o bar.o
  gcc -O2 main.a -o main.exe
  cp main.exe ret0.exe
```

В таком случае скрипт на выходе оставит опции оптимизации:

```
{"type": "compile", "id": "3", "executable": "gcc", "arguments": ["-O1"]}
{"type": "link", "input": ["3"], "executable": "gcc", "arguments": ["-O2"]}
```

2.3.13. Фильтрация по артефактам — `bxy_sieve`

Скрипт позволяет удалить из входного JSON-файла команды, которые не участвуют в создании заданного набора артефактов. Команда считается участвующей в создании артефакта, если артефакт был сгенерирован ей либо если существует цепочка команд, начинающаяся с неё, такая что для каждой пары соседних команд существует файл, который сгенерирован первой и прочитан второй,

причём последняя команда в цепочке генерирует заданный артефакт. По умолчанию также включено добавление в итоговый вывод всех родительских команд участвующих команд.

Один из возможных сценариев применения — исключение команд компиляции и компоновки тестов конфигурации, которые могут повлиять, например, на вывод скрипта `bxy_parse_flags` (см. раздел 2.3.12) таким образом, что в нём появятся нежелательные команды.

```
usage: bxy_sieve [-h] [-a BUILD_ARTIFACT] [-H HASH] [-s SEQNO] [-o OUTPUT]
                 [--hasher HASHER] [--no-keep-parents]
                 json
```

Обязательные аргументы:

- Путь до входного JSON-файла.

Опциональные аргументы:

- С помощью `--build-artifact` можно задать путь к артефакту на машине, на которой запускается этот скрипт. Опция может быть задана произвольное число раз. Скрипт считает его хэш, чтобы определить артефакт в представленной во входном JSON сборке. Эта опция должна использоваться только вместе с `--hasher`.
- С помощью `--hash` можно задать хэш артефакта. Опция может быть задана произвольное число раз.
- С помощью `--seqno` можно задать `seqno` артефакта. Опция может быть задана произвольное число раз.
- `--hasher` позволяет определить командную строку утилиты для подсчёта хэша артефактов, определённых с помощью `--build-artifact`.
- `--no-keep-parents` отключает автоматическое добавление в итоговый вывод предков участвующих в создании артефактов команд.
- С помощью `--output-file` задаётся путь до файла, в который будет записан выходной JSON в формате Buildography. По умолчанию используется стандартный поток вывода.

Если артефакт задан с помощью `--build-artifact` или `--hash`, он не всегда может быть однозначно идентифицирован. В таком случае скрипт завершится с ненулевым кодом возврата, а на стандартный поток ошибок будет выведено сообщение о неоднозначности со списком всех найденных `seqno`, которые соответствуют заданному или вычисленному хэшу.

Если автоматическое добавление родительских команд отключается, то в выходных `component_commands` пересчитываются `parent_id`. Если у команды нет ни одного участвующего в создании артефактов предка, то поле не добавляется в объект команды, в противном случае родителем становится ближайший участвующий предок.

Например, пусть сборка задана с помощью данного `Makefile`:

```
.PHONY: all
all:
  gcc a1.c -c
  gcc a2.c -c
  gcc b1.c -c
  gcc b2.c -c
  gcc c1.c -c
  gcc c2.c -c
  gcc a1.o a2.o -o a.exe
  gcc b1.o b2.o -o b.exe
  gcc c1.o c2.o -o c.exe
```

Предположим, мы хотим оставить в JSON только такие команды, которые участвуют в сборке `a.exe` и `b.exe`. Пусть `a.exe` есть на нашей сборочной машине, а для `b.exe` известно только `seqno`. Тогда запуск

```
bxy_sieve -a /foo/a.exe -s $B_SEQNO build.json -o ab.json
```

оставит в JSON все команды `gcc`, собирающие `a1.o`, `a2.o`, `b1.o`, `b2.o`, `a.exe`, `b.exe`, а также все их команды-потомки, плюс команду `make` как родительскую для `gcc`. Содержимое записей в `component_commands` файла `ab.json` останется таким же, каким и было в `build.json`.

Есть добавить опцию `--no-keep-parents`, то в `ab.json` останутся только команды `cc1`, которые читают `a1.c`, `a2.c`, `b1.c`, `b2.c`; `as`, которые записывают `a1.o`, `a2.o`, `b1.o`, `b2.o`; и `ld`, которые генерируют `a.exe` и `b.exe`.

2.3.14. Проверка соответствия JSON формату Buildography — `bxy_check_json`

Скрипт позволяет проверить соответствие входного JSON-файла схеме, расположенной в `share/bxy-json.schema.yaml`.

```
usage: bxy_check_json [-h] [-q] json
```

Обязательные аргументы:

- Путь до входного JSON-файла.

Опциональные аргументы:

- С помощью `-q` (`--quiet`) можно отключить вывод сообщений о (не)соответствии JSON схеме.

Если входной файл является корректным JSON в формате Buildography, будет напечатано сообщение `JSON is valid`. Иначе скрипт завершится с ненулевым кодом возврата и будет выведено сообщение об ошибке. Вердикт печатается на стандартный поток вывода.

2.4. Проверка сборок на соответствие классу безопасного компилятора

Buildography может использоваться для проверки соответствия трассируемой сборки заданному классу безопасного компилятора.

Типичный сценарий использования состоит из последовательного применения следующих скриптов:

1. `bxy_merge_json` (см. раздел 2.3.5) — для объединения в один JSON-файл результатов трассировки частей одной распределённой сборки или нескольких сборок. Это опциональный шаг.
2. `bxy_sieve` (см. раздел 2.3.13) — для удаления из сборки команд, не принимающих непосредственного участия в сборке, например запусков компилятора на этапе конфигурации. Это опциональный шаг.
3. `bxy_dependency_dag` (см. раздел 2.3.11) — для построения графа зависимостей по данным.
4. `bxy_parse_flags` (см. раздел 2.3.12) — для выделения команд компиляции и компоновки и связей между ними.
5. `test_build` — для запуска квалификационных тестов безопасного компилятора.

Входной формат каждого последующего скрипта является выходным форматом предыдущего. Входной и выходной форматы `bxy_merge_json` и `bxy_sieve`, а также входной формат `bxy_dependency_dag` — это выходной формат Buildography (см. раздел 2.2). `test_build` производит журнал запуска тестов в текстовом формате и генерирует вердикт о прохождении или непрохождении сборкой требований заданного класса безопасного компилятора.

Следующий раздел описывает интерфейс и механизм работы скрипта `test_build`.

2.4.1. Проверка файла с описанием сборки на соответствие классу безопасного компилятора — `test_build`

Скрипт выполняет запуск квалификационных тестов безопасности для файла с описанием сборки проекта.

```
usage: test_build.py [-h] [--level {1,2,3}]
                     [--spec COMPILE_EXE[:LINK_EXE]=SPEC]
                     [--testsuite-dir TESTSUITE_DIR] [--verbose]
                     [BUILD_COMMANDS]
```

- Опция `--level` задаёт тестируемый класс безопасного компилятора, по умолчанию 3.
- Опция `--spec` задаёт файл спецификации для пары компилятор-компоновщик (`COMPILE_EXE`, `LINK_EXE`). Если `LINK_EXE` не задан, файл спецификации задаётся для пары (`COMPILE_EXE`, `COMPILE_EXE`). `COMPILE_EXE` и `LINK_EXE` — это имена команд, и они должны соответствовать командам во входном файле с описанием сборки (см. Раздел 2.4.4). `SPEC` определяет используемый файл спецификации: если путь содержит символ-разделитель путей `'/'`, то значение `SPEC` интерпретируется как путь к файлу спецификации компилятора (см. Раздел 2.4.2), в противном случае `SPEC` — это имя встроенного файла спецификации (см. Раздел 2.4.3).

  Опция может быть задана несколько раз для разных пар компилятор-компоновщик.
- Опция `--testsuite-dir` определяет путь к директории с квалификационными тестами. По умолчанию используется директория данного скрипта.
- Опция `--verbose` включает режим повышенной детализации вывода и может быть задана до двух раз.
- Позиционный аргумент `BUILD_COMMANDS` задаёт имя входного файла с описанием сборки (см. Раздел 2.4.4). По умолчанию используется стандартный поток ввода.

Входной файл с описанием сборки состоит из команд компиляции и компоновки, при этом каждая команда компоновки может быть связана с произвольным числом команд компиляции (через поле `input` — см. Раздел 2.4.4). `test_build` выполняет запуск набора квалификационных тестов для каждой пары из команды компиляции и связанной с ней командой компоновки, во время которого эти команды используются для, соответственно, компиляции и компоновки тестовых файлов. Набор квалификационных тестов завершается успешно, если данная пара команд удовлетворяет требованиям безопасного компилятора заданного класса.

Квалификационные тесты для данного файла с описанием сборки считаются пройденными, если все запуски набора квалификационных тестов завершились успешно.

При этом в режиме по умолчанию вывод скрипта содержит все пары из команды компиляции и команды компоновки, при запуске на которых квалификационные тесты завершились с ошибками, в режиме повышенной детализации (с опцией `--verbose`) — подробный вывод тестов для таких пар, а также список пар, на которых тесты выполнились успешно, в режиме ещё более повышенной детализации (с двумя опциями `--verbose`) — подробный вывод тестов для всех пар, а также все командные строки для повторения индивидуальных запусков набора квалификационных тестов.

2.4.2. Файл спецификации компилятора

Файл спецификации компилятора задаёт параметры запуска спецификационных тестов для заданного компилятора. Файл спецификации представляет собой Bash-скрипт со следующими определениями:

1. `ref`-файл, описывающий различные параметры для тестирования в формате `<ключ>: <значение>`:

ключ	**значение**
`DIAG_ARRAYBOUNDS`	Шаблон диагностического сообщения о выходе за границы массива

ключ	значение
DIAG_DIVBYZERO	Шаблон диагностического сообщения о делении на ноль
DIAG_GETS	Шаблон диагностического сообщения об использовании функции gets
DIAG_SETJMP	Шаблон диагностического сообщения о наличии переменных автоматического класса памяти, которые могли быть изменены между вызовом setjmp и соответствующим вызовом longjmp
DIAG_SHIFTCOUNTNEGATIVE	Шаблон диагностического сообщения о побитовом сдвиге на отрицательное значение
DIAG_SHIFTCOUNTOVERFLOW	Шаблон диагностического сообщения о побитовом сдвиге на значение, превосходящее размер типа
EXIT_FORTIFIED	Код возврата при переполнении буфера при вызове функции стандартной библиотеки
EXIT_PROTECTED	Код возврата при обнаружении записи, выходящей за границу кадра стека
EXIT_SANITIZED	Код возврата при обнаружении неопределенных конструкций посредством механизмов динамического контроля

Определяется переменной cref. Значение — путь к файлу или имя встроенного ref-файла (см. Раздел 2.4.3).

2. Фичи тестирующей системы, задающие различные параметры проверки соответствия классам безопасного компилятора.

фича	значение	описание
arch	Одно из: • ARM • ARM64 • x86 • x86-64	Целевая архитектура
gnu	—	Компилятор поддерживает GNU-расширения C/C++
io	—	Поддерживается работа со стандартными потоками ввода-вывода
null-pointer	Одно из: • baremetal • posix • other	Операционная система: **baremetal** — (нет) **posix** — POSIX-совместимая операционная система **other** — (другой вариант)
fortify-subobject	—	Компилятор поддерживает защиту от выхода за границы ближайшего объемлющего подобъекта при использовании _FORTIFY_SOURCE
fortify3	—	Компилятор поддерживает _FORTIFY_SOURCE=3
stack-protector	—	Механизм обнаружения записи, выходящей за границу кадра стека, технически применим
clobbered	Одно из: • force-volatile • wclobbered • sanitizer	Вариант обработки clobbered-переменных в setjmp/longjmp: **force-volatile** — clobbered-переменные получают квалификатор volatile

фича	значение	описание
		wclobbered — выдаётся предупреждение во время компиляции **sanitizer** — при чтении clobbered-переменных вызывается аварийный останов программы
`array-bounds`	Одно из: • `basic` • `aliases`	Продвинутое обнаружение выхода за границы массива: **basic** — не учитывает алиасы **aliases** — учитывает алиасы
`vfork-shared-memory`	—	При вызове vfork у родителя и ребёнка общая память
`builtin_setjmp`	—	Компилятор поддерживает __builtin_setjmp и __builtin_longjmp

Определяются переменной `features`. Значение — массив фичей. Если у фичи есть значение, то оно в элементе массива `features` отделяется от имени фичи через знак `=`.

2.4.3. Встроенные файлы спецификации компилятора

В поставку инструмента включены следующие файлы спецификации компиляторов и соответствующие им `ref`-файлы:

`safec.spec.sh` и `safec.ref` — файл спецификации и `ref`-файл для Безопасного компилятора «SAFEC»;

`safelang.spec.sh` и `safelang.ref` — для Безопасного компилятора «Safelang»;

`gcc-14.spec.sh` и `gcc-14.ref` — для компилятора GNU GCC версии 14;

`clang-19.spec.sh` и `clang-19.ref` — для компилятора Clang версии 19.

Встроенные файлы спецификации также определяют массивы `safe_flags3`, `safe_flags2` и `safe_flags1` — наборы флагов, с которыми компилятор/компоновщик (флаги нужно включать как в команду компиляции, так и в команду компоновки) соответствует заданному классу безопасного компилятора.

2.4.4. Входной файл с описанием сборки

Файл с описанием сборки представляет собой последовательность команд в формате JSON Lines: каждая строка файла с описанием сборки является валидным JSON-документом, определяющим команду компиляции или команду компоновки.

- Для команды компиляции:

```
{ "type": "compile", "id": <ID>, "executable": <EXECUTABLE>, "arguments":
[<ARGUMENT>...] }
```

- Для команды компоновки:

```
{ "type": "link", "input": [<ID>...], "executable": <EXECUTABLE>, "arguments":
[<ARGUMENT>...] }
```

Здесь:
- `ID` — уникальный идентификатор команды (строка).
- `EXECUTABLE` — имя команды или путь к исполняемому файлу (строка).
- `ARGUMENT` — аргумент команды (строка).

JSON-схема одной строки файла:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "oneOf": [
    {
      "type": "object",
      "properties": {
        "type": { "const": "compile" },
        "id": { "type": "string" },
        "executable": { "type": "string", "minLength": 1 },
        "arguments": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "required": ["type", "id", "executable", "arguments"],
      "additionalProperties": false
    },
    {
      "type": "object",
      "properties": {
        "type": { "const": "link" },
        "input": {
          "type": "array",
          "items": { "type": "string" },
          "minItems": 1
        },
        "executable": { "type": "string", "minLength": 1 },
        "arguments": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "required": ["type", "input", "executable", "arguments"],
      "additionalProperties": false
    }
  ]
}
```

2.5. Поддерживаемые архитектуры и сборочные системы

В настоящее время инструмент работает для AMD64 и ARM64.

Работоспособность протестирована для систем сборки GNU autotools+make, CMake, Meson. Для апробации использовалось программное обеспечение, написанное преимущественно на языках Си и C++.

3. Руководство администратора

3.1. Системные требования

Для использования трассировщика buildography из состава поставки требуются

- ядро Linux версии не ниже 4.8, сконфигурированное с `CONFIG_SECCOMP_FILTER=y`
- glibc версии не ниже 2.27

Средства анализа, входящие в состав поставки, также используют

- bash
- jq
- rhash версии не ниже 1.3.8
- typst (опционально, для генерации файла с зависимостями сборки в формате PDF)

- python версии не ниже 3.10
- pyyaml
- jsonschema (только для `bxy_check_json`)

3.2. Установка предсобранного инструмента

Инструмент распространяется в виде готовых бинарных сборок. Для установки из архива следует использовать команду

```
tar xf buildography-amd64.tar.gz -C <install-dir>
```

4. Принцип работы инструмента

Трассировщик полагается на использование ptrace API — операций, реализованных в ядре Linux и доступных через системный вызов `ptrace` (конкретная операция определяется параметрами вызова).

Существенная деталь реализации трассировщика — использование `seccomp` для фильтрации системных вызовов, при выполнении которых трассировщик получает управление. BPF программа, определяющая, как обрабатывать системные вызовы, выполняется в контексте ядра, что позволяет существенно снизить накладные расходы.

Для получения необходимой информации трассировщик отслеживает следующие системные вызовы:

Системный вызов	Необходимая информация
`open/openat/openat2`	Путь к файлу и режим, в котором он был открыт
`execve`	Список аргументов команды
`fork/vfork/clone`	Идентификатор нового процесса