libqsearch
A library designed for fast multiple pattern matching

Philippe Biondi
<biondi@cartel-securite.fr>

—

FOSDEM 2003
February 8-9th, 2003
What is libqsearch?
- Presentation
- History
- Architecture

Details
- API
- Algorithms
- The whole picture

Other stuff
- Test suite
- Kernel

Conclusion
Outline

What is libqsearch?

- Presentation
- History
- Architecture

Details

- API
- Algorithms
- The whole picture

Other stuff

- Test suite
- Kernel

Conclusion
Lots of programs (IDS, antivirus, ...) need to
  ► search for a lot of patterns at the same time
  ► search in a stream splitted in parts (TCP payloads, split in buffers...)
  ► ...

Algorithms exist but one can’t be good at everything
(long patterns, lots of patterns, regexps, ...)

libqsearch’s aim is to provide many algorithms with one interface

Programmed in C

Patterns can have a type (case (in)sensitive, use jokers, \ldots)

Search for multiple patterns in one call

Types of patterns can be mixed in the same search

Call a callback for each match

Use states to summarize the past of a stream

\begin{itemize}
\item Patterns splitted between 2, 3, \ldots buffers are transparently found
\item States can be saved to carry on a search at a given point of the stream (rollback, TCP packet scanned but finally cancelled by the target, \ldots)
\end{itemize}
Nicolas Brulez (<brulez@cartel-securite.fr>) began to write an open source antivirus.

In complement of his heuristic engine, he needed to search lots of signatures.

I wanted to implement a DFA that would recognize multiple patterns at once (phantasm #7).

The first “fast search experiment” was born: a python program that generated C or ASM code from a list of patterns.

This experiment was submitted to Prelude IDS core team.

Yohann Vandoorselaere convinced me to write a quick search library that could be used by Prelude, and help me build libqsearch API specifications.

libqsearch was born.
Romain Françoise from Arkoon asked me about the portability of libqsearch into Linux kernel

I ported libqsearch API and plugins to the Linux kernel

libqsearch is waiting to be used, and for plugins to be contributed :-(
Program

API
qsearch...
load_plugin
get_algo_list
get_algo
constructor
add_pattern
compile
search
destructor
unload_plugin

Plugin
Algo
Algo

Plugin
Algo
To use libqsearch, we need to manipulate 4 entities

- `qsearch_obj_t`: the search object, which we submit patterns to, compile and search
- `qsearch_algo_t`: the algorithm, which can be instanciated to a search object
- `qsearch_plugin_t`: the plugin, which can contain many algorithms.
- `qsearch_state_t`: the state of a search, that summarize the past of a stream
Plugin name

load_plugin()

get_algo_list()

Plugin

unload_plugin()

get_algo()

constructor()

Pattern

add_pattern()

Search obj

Search obj

destructor()

Buffer

compile()

Compiled search obj

State

search()

free_state()
What is it?

**buffer 1**

state0 = NULL

search()

**buffer 2**

state 1

search()

**buffer 3**

state 2

search()

state 3
state0 = NULL

buffer 1

search()

state 1

buffer 2

search()

state 2

buffer 3

search()

state 3

buffer 3'

search()
What is libqsearch?
- Presentation
- History
- Architecture

Details
- API
- Algorithms
- The whole picture

Other stuff
- Test suite
- Kernel

Conclusion
To do a search:

- Load the plugin, given its name (qsearch_load_plugin())
- Get an algo from the plugin, given its name (qsearch_get_algo())
- Instantiate a search object from the algo (qsearch_constructor())
- Add patterns to the search object (qsearch_add_pattern() for each pattern)
- Compile the set of patterns (qsearch_compile())
- Look for the patterns in buffers (qsearch_search())
- Destruct the search_object (qsearch_destructor())
- Unload the plugin (qsearch_unload_plugin())
qsearch_plugin_t *qsearch_load_plugin(const char *name);

- This function look for a plugin from its name

- user space implementation: search it in the plugin directory if <name> does not contain any ’/’

- kernel space implementation: if not already registered, try to load the module qsearch_name

- returns a (qsearch_plugin_t *) or NULL if there was an error.
int qsearch_unload_plugin(qsearch_plugin_t *plugin);

- This function will unload a previously loaded plugin
- Don’t forget to free each search object and state related to algorithm from this plugin
qsearch_algo_t **qsearch_get_algo_list(
    qsearch_plugin_t *plugin);

- returns a NULL terminated list of algorithms provided by the plugin
qsearch_algo_t *qsearch_get_algo(qsearch_plugin_t *plugin, char *name);

- returns an algorithm provided by the plugin, being given the algorithm name
- returns NULL if the plugin does not contain such an algorithm.
qsearch_obj_t *qsearch_constructor(qsearch_algo_t *algo, qsearch_options_t *options);

- instantiate an algorithm.
- you can instantiate as many search objects as you want from the same algorithm
- the options are not used yet
- returns NULL if error
int qsearch_destructor(qsearch_obj_t *self);

- destructs a previously instantiated search object
- returns non-zero if error
int qsearch_add_pattern(qsearch_obj_t *self, char *pattern, int len, void *ptrn_data, int type);

- Adds a pattern for it to be looked for later

- Each pattern must be given a type
  - There are for the moment 5 types:
    - #define QSEARCH_CS 0 /* normal case sensitive pattern */
    - #define QSEARCH_CI 1 /* case insensitive pattern */
    - #define QSEARCH_JOKER_CS 2 /* pattern using '*' for any char */
    - #define QSEARCH_JOKER_CI 3 /* pattern using '*' for any char */
    - #define QSEARCH_REGEXP 4 /* regular expression */

- An algorithm may or may not support a given type

- If not supported, returns -QEBADTYPE.

- Per-pattern data pointer: each time a pattern matches, a callback will be called with the related ptrn_data.

- Returns non-zero if error.
int qsearch_compile(qsearch_obj_t *self);

- patterns can’t be looked for before.
- once every patterns have been added, the search object can be compiled
- no more patterns can be added after that
- Returns non-zero if error.
int qsearch_search(qsearch_obj_t *self,
                   qsearch_state_t *state_in,
                   qsearch_state_t **state_out,
                   qsearch_state_t *state_io,
                   qsearch_callback_t cb, void *cb_data,
                   void *buffer, size_t len);

- Once every patterns are added and the search object is compiled, we can look for patterns in a buffer.

- If an algo does not implement the states management, qsearch_search() will return an error (-QESTATES) if state_in or state_out are not NULL before doing any search.
The search can be stateful. 2 ways to do that:

► **use** `state_in` and `state_out`, `state_io` is always NULL.
  - the function will put a summary of the past search session in `state_out`
  - `state_out` will be used as `state_in` in the next call
  - the very first search will be given NULL as `state_in`
  - memory used by `state_out` is allocated by `qsearch_search()`, but you have to free it with `qsearch_free_state()`
  - `state_in` is neither modified nor destructed

► allocate a `state_io` variable with `qsearch_alloc_io_state()`
  - it will be as both `state_in` and `state_out`
  - it can save the time of malloc/free
  - you won’t be able to replay searches
typedef int (*qsearch_callback_t)(void *cb_data, void *ptrn_data, size_t match_offset);

- When a match occurs: the provided callback will be called with
  - the position of the match
  - the per-pattern data pointer (qsearch_add_pattern())
  - the call-back data pointer (qsearch_search()).
- If the search is stateful, the match position is absolute
- else, it’s relative to the beginning of the current buffer
- the position of the match points the end of the pattern.
  position of "abc" in "abcd" will be 3.
- the callback function return
  - 0 ➝ the search will carry on
  - other ➝ the search ends, the return value is returned by `qsearch_search()`.

- if `qsearch_search()` raises an error, it will return a negative value.

- to distinguish callback return value and `qsearch_search()` return value, the callback should only return positive values.
int qsearch_free_state(qsearch_obj_t *self,
qsearch_state_t *state);

- free a state allocated by qsearch_search().
int qsearch_check_pattern_type(qsearch_obj_t *self, int type);

- check if a pattern type is supported by a search object
- return 0 if the pattern type is supported
To write a new plugin

- you can start with the skeleton plugin
- you have to follow some rules
Rules

- You SHOULD support both states mechanisms.
- If `state_io not supported`, `alloc_io_state()` must return `-QESTATES`.
- If supported, position of match MUST be absolute from the beginning of the stream.
- If not supported you MUST return `-QESTATES` when searching with non NULL `state_in|out|io`.

- The position of a match MUST point to the end of the pattern (ex: the position of "abc" in "abcd" is 3).
- You SHOULD support as many types as possible.
- You MUST permit to mix the pattern types you support.
- You MUST NOT use `#include` directive.
- You MUST NOT use external libs not allowed by libqsearch.
Kind of object programming

- Algorithms are classes

- Search objects are instances of algorithms

- `add_pattern()`, `compile()`, `search()`, ..., are methods of search objects
int qsearch_compile(qsearch_obj_t *self)
{
    return (*self->statics->compile)(self);
}

int qsearch_alloc_io_state(qsearch_obj_t *self, qsearch_state_t **state)
{
    return (*self->statics->alloc_io_state)(self, state);
}

int qsearch_free_state(qsearch_obj_t *self, qsearch_state_t **state)
{
    return (*self->statics->free_state)(self, state);
}

int qsearch_destructor(qsearch_obj_t *self)
{
    return (*self->statics->destructor)(self);
}
static skeleton_algo_t algo_skeleton = {
    "skeleton",
    "This is an example",
    1<<QSEARCH_CS | 1<<QSEARCH_CI | 1<<QSEARCH_JOKER_CS | 1<<QSEARCH_JOKER_CI,
    &skeleton_constructor,
    &skeleton_destructor,
    &skeleton_add_pattern,
    &skeleton_compile,
    &skeleton_search,
    &skeleton_alloc_io_state,
    &skeleton_free_state,
};

skeleton_algo_t *QSEARCH_ALGO_LIST_SYM[] = {
    &algo_skeleton,
    NULL
};
What is libqsearch?
  - Presentation
  - History
  - Architecture

Details
  - API
  - Algorithms
  - The whole picture

Other stuff
  - Test suite
  - Kernel

Conclusion
The test suite program is used to
- test the accuracy of algorithms
- in the future: do speed benchmarks

The test suite program loads the specified plugin, and tests the specified (or all) algo

Each accuracy test provides
- patterns to be looked for (with their type)
- a test sample in which patterns can be looked for
- for each pattern: where it should have been found in the sample
The config file contains the list of tests

```plaintext
accuracy "ci" {
    info "one pattern case insensitive search";
    patterns "abcd":CI;
    sample "..abcd..ABCD..AbCd..aBcD..";
    results "000010000010000010000100001000100";
};

accuracy "mixed" {
    info "mixed pattern types";
    patterns {
        "abc":CS;
        "abcd":CI;
    };
    sample "..abcd..ABCD..AbCd..abcd..";
    results { "00001000000000000000100000001000";
           "000001000001000001000010000100";
    };
};
```
For each accuracy test, the search object is fed with the sample been splitted up in 1 octet pieces, then in 2 octets, \ldots, then with the whole sample at once.

This enables the detection of some hard off-by-one bugs in algorithms implementations.
Kernelize script

- apply a small patch
  - to modify main Makefile
  - to include new compilation menu options
  - to add an init call in misc devices (for the test module)

- copy the qsearch directory (API kernel implementation) into the kernel dir

- for each plugin
  - copy the plugin
  - generate a wrapper and a Makefile
  - add a line in the config menu

- generate the qsearch main Makefile
Result:

<M> Kernel QSearch support
--- Algorithms
<M> skeleton
<M> simple
<M> bm
--- Tools
<M> QSearch test module
#define PLUGIN "@PLUGIN_NAME@"

#define malloc(x) kmalloc(x, GFP_KERNEL)
#define free(x) kfree(x)

[...]

#include PLUGIN ".c"
[...]

MODULE_AUTHOR("Philippe Biondi <biondi@cartel_securite.fr>");
MODULE_DESCRIPTION("libqsearch kernel wrapper for " PLUGIN " plugin");
MODULE_LICENSE("GPL");

[...]

#ifdef MODULE
int init_module(void)
[...]
void cleanup_module(void)
[...]

What is libqsearch?
- Presentation
- History
- Architecture

Details
- API
- Algorithms
- The whole picture

Other stuff
- Test suite
- Kernel

Conclusion
libqsearch is fully fonctionnal

3 plugins:
  - skeleton (does not find anything)
  - simple one
  - Boyer-Moore-like with CS/CI/JokerCS/JokerCI support

a test suite

fully fonctionnal in kernel space

a test module and tools to check the API from user space

very good performances
That’s all folks. Thanks for your attention.

You can reach me at <phil@lids.org>

These slides are available at
http://www.cartel-securite.fr/pbiondi/

libqsearch is available at
http://www.cartel-securite.fr/pbiondi/libqsearch.html
or

cvs
-d:pserver:anonymous@cvs.prelude-ids.org:/cvsroot/prelude
co libqsearch