HawkTracer profiler

Marcin Kolny

Amazon Prime Video
marcin.kolny@gmail.com

February 2, 2020

Why do we need another profiler?

Environment:

- Limited access to the device
- Lack of development tools
- Various low-end platforms
- Various languages (C++ for native, Lua and JavaScript for scripted)

Why do we need another profiler?

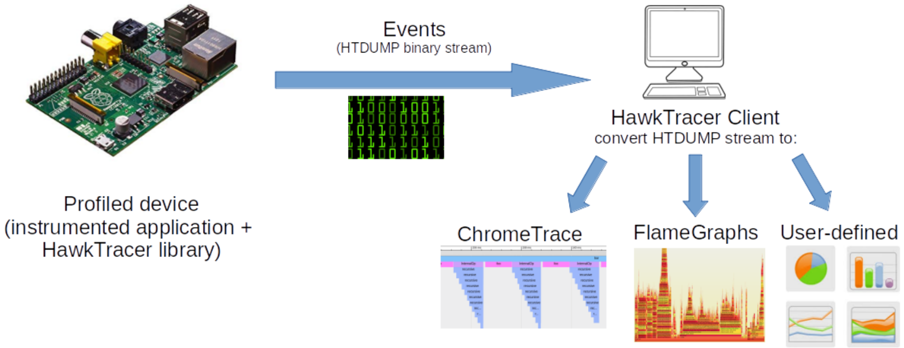
Environment:

- Limited access to the device
- Lack of development tools
- Various low-end platforms
- Various languages (C++ for native, Lua and JavaScript for scripted)

HawkTracer features:

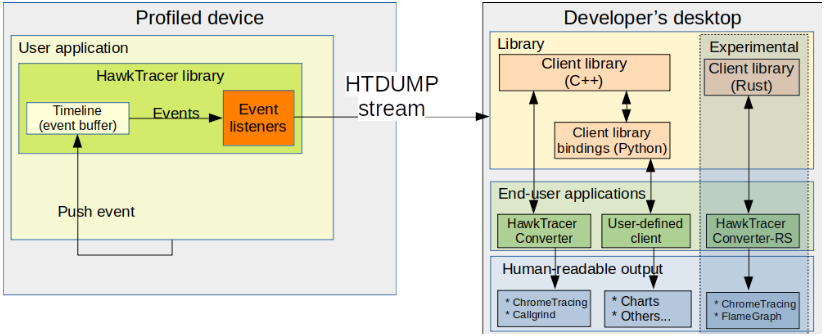
- User space & instrumentation based
- Written in C (and C++) but available for other languages
- Built-in to executable as a library ("install app" only)
- Low cost of porting (to SmartTVs/Consoles/Streaming Sticks/...)
- Measure timings as well as arbitrary resource usage
- Low overhead (lock-free when possible)
- Consistent user experience across all supported platforms

High Level architecture



- Event - base data unit (predefined or user-defined event types)
- HTDUMP stream - binary stream (sent to a client over TCP / File / user-defined protocol)
- Client - converts HTDUMP stream to human-readable representation

Data flow / component diagram



- Timeline - event buffer, lock-free or thread-safe (up to the usecase)
- Event Listener - processes batch of events (e.g. store to file, send over TCP/IP)
- Client library - converts HTDUMP stream to list of Event structures

Global Timeline

- predefined in the HawkTracer library
- recommended for most of the usecases
- per-thread instance (no locks required)
- `ht_global_timeline_get()`

Defining event types

- C structure with arbitrary fields
- support for inheritance
- runtime structure introspection (using MKCREFLECT library)

```
HT_DECLARE_EVENT_KLASS(
  MyEvent, // Event class name
  HT_Event, // Base event
  (INTEGER, uint8_t, field_1), // field definition (type, C type, field name)
  (STRING, char*, field_2)     // field definition (type, C type, field name)
  // Other fields...
)
```

Converts to C structure and a few helper methods:

```
typedef struct {
  HT_Event base;
  uint8_t field_1;
  char* field_2;
} MyEvent;

// Serializes event to HTDUMP format
size_t ht_MyEvent_fnc_serialize(
  HT_Event* event, HT_Byte* buffer);
```

```
typedef struct {
  HT_EventKlass* klass;
  uint64_t timestamp_ns;
  uint64_t event_id;
} HT_Event;

// Information about the class structure
MKCREFLECT_TypeInfo*
  mkcreflect_get_MyEvent_type_info(void);
```

Pushing event to a timeline:

```
HT_TIMELINE_PUSH_EVENT(timeline, MyEvent, 28, "Hello␣World!");
```

HTDUMP Event stream

- **Metadata stream** - information about event types
(transferred as `HT_EventKlassInfoEvent` and `HT_EventKlassFieldInfoEvent` events)

```
HT_EventKlassInfoEvent {                          //     33 bytes
    "type": U32(2)                                // 02 00 00 00
    "timestamp": U64(394021837478301)             // 9D 19 A8 5B 5C 66 01 00
    "id": U64(38)                                 // 26 00 00 00 00 00 00 00
    "info_klass_id": U32(9)                       // 09 00 00 00
    "event_klass_name": Str("MyEvent")            // 4D 79 45 76 65 6E 74 00
    "field_count": U8(3)                          // 03
}
HT_EventKlassFieldInfoEvent {                     //     49 bytes
    "type": U32(3)                                // 03 00 00 00
    "timestamp": U64(394021837479489)             // 41 1E A8 5B 5C 66 01 00
    "id": U64(40)                                 // 28 00 00 00 00 00 00 00
    "info_klass_id": U32(9)                       // 09 00 00 00
    "field_type": Str("uint8_t")                  // 75 69 6E 74 38 5F 74 00
    "field_name": Str("field_1")                  // 66 69 65 6C 64 5F 31 00
    "size": U64(1)                                // 01 00 00 00 00 00 00 00
    "data_type": U8(99)                           // 63
}
// ...
```

- Events stream

```
MyEvent {                                         //     34 bytes
  "type": U32(9)                                  // 09 00 00 00
  "timestamp": U64(394021837504177)               // B1 7E A8 5B 5C 66 01 00
  "id": U64(42)                                   // 2A 00 00 00 00 00 00 00
  "field_1": U8(28)                               // 1C
  "field_2": Str("Hello␣World!")                  // 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00
}
```

Measuring time - predefined events

- C / C++

```
void foo()
{
  HT_TRACE_FUNCTION(timeline);
  // HT_G_TRACE_FUNCTION() for Global Timeline
  // ...
  { // new scope
    HT_TRACE(timeline, "custom␣label");
    // HT_G_TRACE("custom label") for Global Timeline
    // use HT_TRACE_OPT_* for better performance
  }
}
```

- Python

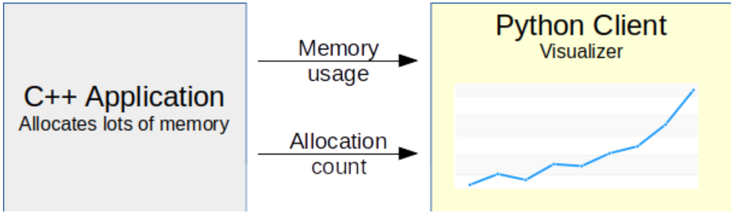
```
from hawktracer.core import trace

@trace  # uses Global Timeline
def foo():
  pass
```

- Rust

```
#[hawktracer(trace_this)] // uses Global Timeline
fn method_to_trace() {
  // ...
  { // new scope
    scoped_tracepoint!(_custom_label);
    // ...
  }
}
```

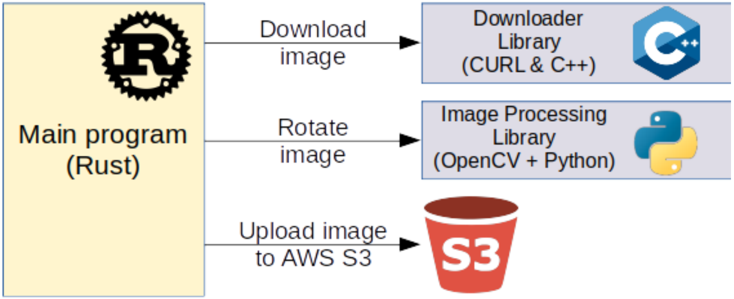
Demo - Real-time data stream

Writing custom client



Demo - Cross-language project

Rust & Python & C



Future improvements

- Generic data viewer
- CTF support
- Bindings for more languages (JavaScript)
- Allow custom event type definitions from bindings
- ...

Thank you!

- marcin.kolny@gmail.com
- HawkTracer website:
  (entry point, community, how to get involved)
  www.hawktracer.org
- Documentation:
  (reference, tutorials, design concepts, integration)
  www.hawktracer.org/doc
- Code repository:
  - HawkTracer Core:
    github.com/amzn/hawktracer
  - HawkTracer Converter (Rust):
    github.com/loganek/hawktracer-converter
  - HawkTracer Rust bindings:
    github.com/AlexEne/rust_hawktracer