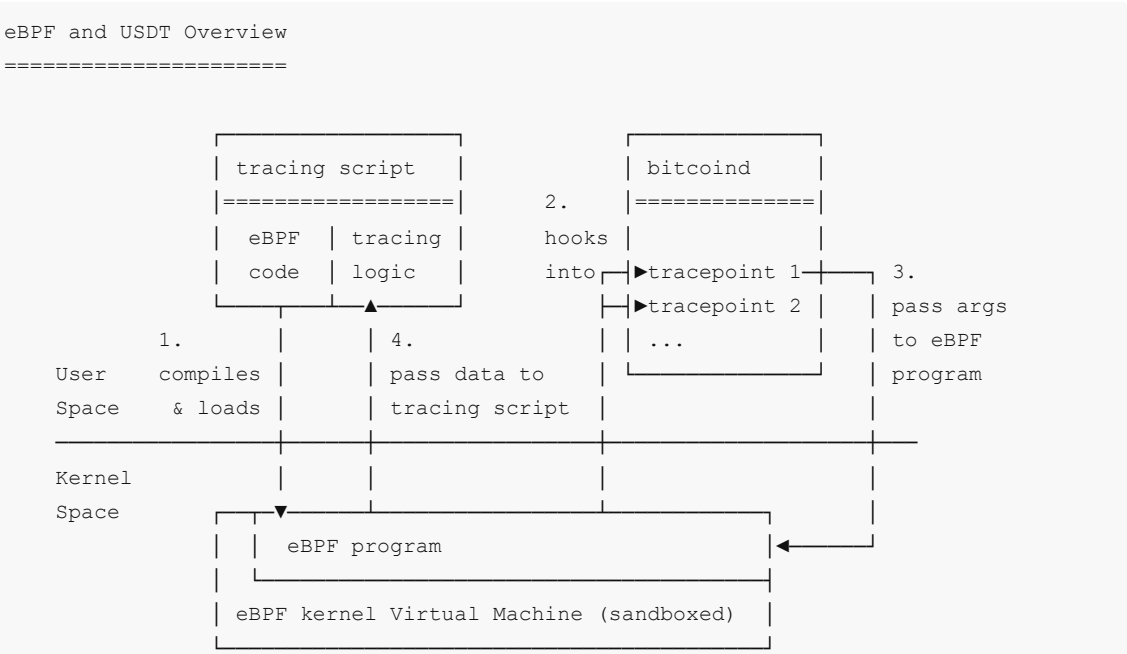
User-space, Statically Defined Tracing (USDT) for Bitcoin Core

Bitcoin Core includes statically defined tracepoints to allow for more observability during development, debugging, code review, and production usage. These tracepoints make it possible to keep track of custom statistics and enable detailed monitoring of otherwise hidden internals. They have little to no performance impact when unused.



The Linux kernel can hook into the tracepoints during runtime and pass data to sandboxed eBPF programs running in the kernel. These eBPF programs can, for example, collect statistics or pass data back to user-space scripts for further processing.

The two main eBPF front-ends with support for USDT are bpftrace and BPF Compiler Collection (BCC). BCC is used for complex tools and daemons and `bpftrace` is preferred for one-liners and shorter scripts. Examples for both can be found in contrib/tracing.

Tracepoint documentation

The currently available tracepoints are listed here.

Context `net`

Tracepoint `net:inbound_message`

Is called when a message is received from a peer over the P2P network. Passes information about our peer, the connection and the message as arguments.

Arguments passed:

1. Peer ID as `int64`
2. Peer Address and Port (IPv4, IPv6, Tor v3, I2P, ...) as `pointer to C-style String` (max. length 68 characters)
3. Connection Type (inbound, feeler, outbound-full-relay, ...) as `pointer to C-style String` (max. length 20 characters)
4. Message Type (inv, ping, getdata, addrv2, ...) as `pointer to C-style String` (max. length 20 characters)
5. Message Size in bytes as `uint64`
6. Message Bytes as `pointer to unsigned chars` (i.e. bytes)

Note: The message is passed to the tracepoint in full, however, due to space limitations in the eBPF kernel VM it might not be possible to pass the message to user-space in full. Messages longer than a 32kb might be cut off. This can be detected in tracing scripts by comparing the message size to the length of the passed message.

Tracepoint `net:outbound_message`

Is called when a message is send to a peer over the P2P network. Passes information about our peer, the connection and the message as arguments.

Arguments passed:

1. Peer ID as `int64`
2. Peer Address and Port (IPv4, IPv6, Tor v3, I2P, ...) as `pointer to C-style String` (max. length 68 characters)
3. Connection Type (inbound, feeler, outbound-full-relay, ...) as `pointer to C-style String` (max. length 20 characters)
4. Message Type (inv, ping, getdata, addrv2, ...) as `pointer to C-style String` (max. length 20 characters)
5. Message Size in bytes as `uint64`
6. Message Bytes as `pointer to unsigned chars` (i.e. bytes)

Note: The message is passed to the tracepoint in full, however, due to space limitations in the eBPF kernel VM it might not be possible to pass the message to user-space in full. Messages longer than a 32kb might be cut off. This can be detected in tracing scripts by comparing the message size to the length of the passed message.

Context `validation`

Tracepoint `validation:block_connected`

Is called *after* a block is connected to the chain. Can, for example, be used to benchmark block connections together with `-reindex`.

Arguments passed:

1. Block Header Hash as `pointer to unsigned chars` (i.e. 32 bytes in little-endian)
2. Block Height as `int32`
3. Transactions in the Block as `uint64`
4. Inputs spend in the Block as `int32`
5. SigOps in the Block (excluding coinbase SigOps) `uint64`
6. Time it took to connect the Block in microseconds (µs) as `uint64`

Context `utxocache`

The following tracepoints cover the in-memory UTXO cache. UTXOs are, for example, added to and removed (spent) from the cache when we connect a new block. **Note**: Bitcoin Core uses temporary clones of the *main* UTXO cache (`chainstate.CoinsTip()`). For example, the RPCs `generateblock` and `getblocktemplate` call `TestBlockValidity()` , which applies the UTXO set changes to a temporary cache. Similarly, mempool consistency checks, which are frequent on regtest, also apply the the UTXO set changes to a temporary cache. Changes to the *main* UTXO cache and to temporary caches trigger the tracepoints. We can't tell if a temporary cache or the *main* cache was changed.

Tracepoint `utxocache:flush`

Is called *after* the in-memory UTXO cache is flushed.

Arguments passed:

1. Time it took to flush the cache microseconds as `int64`
2. Flush state mode as `uint32` . It's an enumerator class with values `0` (`NONE`), `1` (`IF_NEEDED`), `2` (`PERIODIC`), `3` (`ALWAYS`)
3. Cache size (number of coins) before the flush as `uint64`
4. Cache memory usage in bytes as `uint64`
5. If pruning caused the flush as `bool`

Tracepoint `utxocache:add`

Is called when a coin is added to a UTXO cache. This can be a temporary UTXO cache too.

Arguments passed:

1. Transaction ID (hash) as `pointer to unsigned chars` (i.e. 32 bytes in little-endian)
2. Output index as `uint32`
3. Block height the coin was added to the UTXO-set as `uint32`
4. Value of the coin as `int64`
5. If the coin is a coinbase as `bool`

Tracepoint `utxocache:spent`

Is called when a coin is spent from a UTXO cache. This can be a temporary UTXO cache too.

Arguments passed:

1. Transaction ID (hash) as `pointer to unsigned chars` (i.e. 32 bytes in little-endian)
2. Output index as `uint32`
3. Block height the coin was spent, as `uint32`
4. Value of the coin as `int64`
5. If the coin is a coinbase as `bool`

Tracepoint `utxocache:uncache`

Is called when a coin is purposefully unloaded from a UTXO cache. This happens, for example, when we load an UTXO into a cache when trying to accept a transaction that turns out to be invalid. The loaded UTXO is uncached to avoid filling our UTXO cache up with irrelevant UTXOs.

Arguments passed:

1. Transaction ID (hash) as `pointer to unsigned chars` (i.e. 32 bytes in little-endian)
2. Output index as `uint32`
3. Block height the coin was uncached, as `uint32`
4. Value of the coin as `int64`
5. If the coin is a coinbase as `bool`

Adding tracepoints to Bitcoin Core

To add a new tracepoint, `#include <util/trace.h>` in the compilation unit where the tracepoint is inserted. Use one of the `TRACEx` macros listed below depending on the number of arguments passed to the tracepoint. Up to 12 arguments can be provided. The `context` and `event` specify the names by which the tracepoint is referred to. Please use `snake_case` and try to make sure that the tracepoint names make sense even without detailed knowledge of the implementation details. Do not forget to update the tracepoint list in this document.

```
#define TRACE(context, event)
#define TRACE1(context, event, a)
#define TRACE2(context, event, a, b)
#define TRACE3(context, event, a, b, c)
#define TRACE4(context, event, a, b, c, d)
#define TRACE5(context, event, a, b, c, d, e)
#define TRACE6(context, event, a, b, c, d, e, f)
#define TRACE7(context, event, a, b, c, d, e, f, g)
#define TRACE8(context, event, a, b, c, d, e, f, g, h)
#define TRACE9(context, event, a, b, c, d, e, f, g, h, i)
#define TRACE10(context, event, a, b, c, d, e, f, g, h, i, j)
#define TRACE11(context, event, a, b, c, d, e, f, g, h, i, j, k)
#define TRACE12(context, event, a, b, c, d, e, f, g, h, i, j, k, l)
```

For example:

```
TRACE6(net, inbound_message,
    pnode->GetId(),
    pnode->m_addr_name.c_str(),
    pnode->ConnectionTypeAsString().c_str(),
    sanitizedType.c_str(),
    msg.data.size(),
    msg.data.data()
);
```

Guidelines and best practices

Clear motivation and use-case

Tracepoints need a clear motivation and use-case. The motivation should outweigh the impact on, for example, code readability. There is no point in adding tracepoints that don't end up being used.

Provide an example

When adding a new tracepoint, provide an example. Examples can show the use case and help reviewers testing that the tracepoint works as intended. The examples can be kept simple but should give others a starting point when working with the tracepoint. See existing examples in contrib/tracing/.

No expensive computations for tracepoints

Data passed to the tracepoint should be inexpensive to compute. Although the tracepoint itself only has overhead when enabled, the code to compute arguments is always run - even if the tracepoint is not used. For example, avoid serialization and parsing.

Semi-stable API

Tracepoints should have a semi-stable API. Users should be able to rely on the tracepoints for scripting. This means tracepoints need to be documented, and the argument order ideally should not change. If there is an important reason to change argument order, make sure to document the change and update the examples using the tracepoint.

eBPF Virtual Machine limits

Keep the eBPF Virtual Machine limits in mind. eBPF programs receiving data from the tracepoints run in a sandboxed Linux kernel VM. This VM has a limited stack size of 512 bytes. Check if it makes sense to pass larger amounts of data, for example, with a tracing script that can handle the passed data.

`bpftrace` argument limit

While tracepoints can have up to 12 arguments, bpftrace scripts currently only support reading from the first six arguments (`arg0` till `arg5`) on `x86_64` . bpftrace currently lacks real support for handling and printing binary data, like block header hashes and txids. When a tracepoint passes more than six arguments, then string and integer arguments should preferably be placed in the first six argument fields. Binary data can be placed in later arguments. The BCC supports reading from all 12 arguments.

Strings as C-style String

Generally, strings should be passed into the `TRACEx` macros as pointers to C-style strings (a null-terminated sequence of characters). For C++ `std::strings` , `c_str()` can be used. It's recommended to document the maximum expected string size if known.

Listing available tracepoints

Multiple tools can list the available tracepoints in a `bitcoind` binary with USDT support.

GDB - GNU Project Debugger

To list probes in Bitcoin Core, use `info probes` in `gdb` :

```
$ gdb ./src/bitcoind
…
(gdb) info probes
Type Provider   Name             Where              Semaphore Object
stap net        inbound_message  0x000000000014419e /src/bitcoind
stap net        outbound_message 0x0000000000107c05 /src/bitcoind
stap validation block_connected  0x00000000002fb10c /src/bitcoind
…
```

With `readelf`

The `readelf` tool can be used to display the USDT tracepoints in Bitcoin Core. Look for the notes with the description `NT_STAPSDT` .

```
$ readelf -n ./src/bitcoind | grep NT_STAPSDT -A 4 -B 2
Displaying notes found in: .note.stapsdt
  Owner                 Data size       Description
  stapsdt              0x0000005d       NT_STAPSDT (SystemTap probe descriptors)
    Provider: net
    Name: outbound_message
    Location: 0x0000000000107c05, Base: 0x0000000000579c90, Semaphore:
0x0000000000000000
    Arguments: -8@%r12 8@%rbx 8@%rdi 8@192(%rsp) 8@%rax 8@%rdx
…
```

With `tplist`

The `tplist` tool is provided by BCC (see Installing BCC). It displays kernel tracepoints or USDT probes and their formats (for more information, see the `tplist` usage demonstration). There are slight binary naming differences between distributions. For example, on Ubuntu the binary is called `tplist-bpfcc`.

```
$ tplist -l ./src/bitcoind -v
b'net':b'outbound_message' [sema 0x0]
  1 location(s)
  6 argument(s)
…
```