TEE-Perf: A Profiler for Trusted Execution Environments

Maurice Bailleu[†], Donald Dragoti[‡], Pramod Bhatotia[†], Christof Fetzer[‡]
[†]The University of Edinburgh [‡] TU Dresden

***Abstract*—We introduce TEE-PERF, an architecture- and platform-independent performance measurement tool for trusted execution environments (TEEs). More specifically, TEE-PERF supports method-level profiling for unmodified multithreaded applications, without relying on any architecture-specific hardware features (e.g. Intel VTune Amplifier), or without requiring platform-dependent kernel features (e.g. Linux `perf`). Moreover, TEE-PERF provides accurate profiling measurements since it traces the entire process execution without employing instruction pointer sampling. Thus, TEE-PERF does not suffer from sampling frequency bias, which can occur with threads scheduled to align to the sampling frequency.**

We have implemented TEE-PERF with an easy to use interface, and integrated it with Flame Graphs to visualize the performance bottlenecks. We have evaluated TEE-PERF based on the Phoenix multithreaded benchmark suite and real-world applications (RocksDB, SPDK, etc.), and compared it with Linux `perf`. Our experimental evaluation shows that TEE-PERF incurs low profiling overheads, while providing accurate profile measurements to identify and optimize the application bottlenecks in the context of TEEs. TEE-PERF is publicly available.

I. INTRODUCTION

Hardware-assisted trusted execution environments (TEEs), such as ARM TrustZone [9], Intel SGX [13], AMD SEV [8] and RISC-V Keystone [26], provide an appealing way to build secure applications for the untrusted computing environment. In particular, TEEs are increasingly being used in the context of *shielded execution* to build a wide-range of secure applications [10, 12, 20, 24, 28, 33]. Shielded execution aims to provide strong confidentiality and integrity properties using a hardware-protected secure memory region.

However, developing applications for TEEs is quite challenging. While building a secure application, the application programmer not only has to preserve the confidentiality and integrity guarantees provided by the TEE, but (s)he also needs to ensure that the application achieves high performance.

Unfortunately, application performance profiling, i.e., to understand the performance bottlenecks inside the TEEs is quite difficult [16]. This is due to the fact that the application performance significantly varies inside the TEE due to the micro-architectural implementation details of the secure hardware. For instance, the cost of random memory accesses in TEEs significantly increases due to the memory encryption engine, which usually operates at the granularity of cache lines. Likewise, the cost of accessing memory beyond the secure physical memory region (allocated in the main memory) incurs very high performance overheads due to secure paging; for example, the Intel SGX architecture supports EPC paging, a mechanism to securely swap enclave pages to unprotected host memory that can slow down application performance up to 2000$\times$ [10]. In addition, the application suffers significant performance cost when performing a context switch from the normal world to the secure world of the TEE since the hardware needs to ensure that the context switch does not leak any information stored in the TEE, e.g. flushing or restoring the translation lookaside buffer (TLB). Further, direct I/O is forbidden inside TEEs, and therefore, the I/O operations have to pass through some wrappers resulting in different performance characteristics as a developer might expect.

To summarize, the micro-architectural implementation of TEEs provides significant challenges for profiling applications while designing high performance applications. Further, many applications need to be profiled across different TEE platforms since they are designed to operate across multiple platforms and architectures. However, there is very little support for performance debugging for TEE, as the conventional profiling tools are either tightly coupled to specific architectures or operating systems. For instance, Intel VTune Amplifier profiler [5] is a proprietary profiler that is specifically designed for the Intel architecture. `perf` [6] relies on the Linux kernel infrastructure for the application profiling. SGX-perf [34] targets Intel SGX architecture specifically, and does not provide method-level application profiling information.

To overcome the limitations of the existing profilers, we propose TEE-PERF, a performance profiling tool for TEEs. More specifically, TEE-PERF targets three design points:

- *Generality:* TEE-PERF provides an architecture- and platform-independent profiling infrastructure.
- *Transparency:* TEE-PERF supports unmodified multi-threaded applications with an easy-to-use interface.
- *Accuracy:* TEE-PERF provides accurate method-level profiling, without resorting to the instruction sampling.

At a high-level, TEE-PERF is based on four straightforward stages (see Figure 1). Firstly, the application is recompiled using our compiler pass, which is used to transparently inject profiling code in the application at the call and return instructions. In the second stage, the recorder collects the performance characteristics while the application is running inside the TEE. The recorder relies on a (hardware- and platform-independent) software counter to capture the application performance profile. Next, the analyzer dissects the recorded log to accurately report the method-level performance characteristics. Lastly, we integrated TEE-PERF with Flame Graphs [1] to visualize the bottlenecks.

We have implemented the TEE-PERF tool with an easy-to-use interface for the application programmers. In addition, TEE-PERF provides several additional design features: support for multi-threaded applications, call stack profiling, a rich

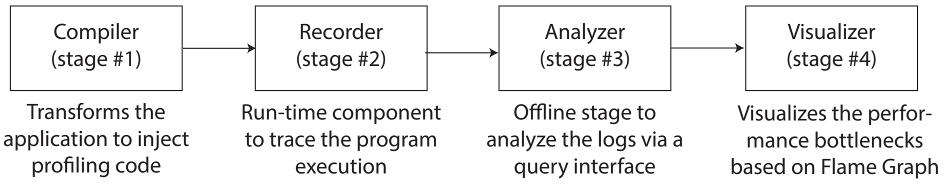
Compiler
(stage #1)
Transforms the application to inject profiling code
Recorder
(stage #2)
Run-time component to trace the program execution
Analyzer
(stage #3)
Offline stage to analyze the logs via a query interface
Visualizer
(stage #4)
Visualizes the performance bottlenecks based on Flame Graph

Figure 1: TEE-PERF overview

declarative query interface, and selective code profiling.

We have experimentally evaluated TEE-PERF based on the Phoenix multithreaded benchmark suite and real-world applications. Our experimental evaluation shows that TEE-PERF incurs low profiling overheads, while providing accurate profile measurements compared with Linux `perf`. Furthermore, we showcase that TEE-PERF is able to identify and optimize the application bottlenecks in the context of TEEs. In particular, we have successfully used the tool to increase the performance to near native performance of Intel's SPDK [18] running inside a SGX enclave.

II. DESIGN

A. Overview

In this section, we present the design of TEE-PERF.

Design goals and assumptions. The primary goal of TEE-PERF is to build a performance measurement tool, which is independent of the underlying operating systems and architectures. Therefore, the tool should also be able to make these measurements independent of the TEE implementation; i.e., different instruction sets (x86 or RISC) or versions (SGX v1 or SGX v2). Since it is platform independent, we do not require any performance counters or timer being available to make introspection into the TEE. Further, we aim to support unmodified multi-threaded applications with an easy-to-use interface. TEE-PERF provides accurate time measurements statistics at function level, which enables applications programmer to identify performance bottlenecks in the context of TEEs. However, we assume that the application running inside TEE is able to access the shared memory with a profiling application (or the recorder) running natively on the host. Additionally, the operating system must support multitasking, i.e., it is able to run a process in the TEE and a process natively outside in parallel. These assumptions hold for most commercially available TEEs and operating systems. We also think that these assumptions will also be valid for future TEEs, which might have additional security features. Lastly, we note that TEE-PERF is designed to be used in the development and debugging phases. Therefore, we can accept the performance overhead of an architecture independent software counter.

Design overview. Figure 1 shows the high-level architecture of TEE-PERF. The tool consists of four main phases: (a) compiler, (b) recorder, (c) analyzer, and (d) visualizer. The compiler pass transforms the application to inject the profiling code in the executable. The recorder sets up the application in the TEE, while simultaneously initiating a process in parallel, which runs natively outside the TEE. The recorder also establishes the shared memory communication medium between the two processes, and it maps a fast reasonable accurate software counter into the TEE.

After the measurement phase, the analyzer dissects the performance measurement log file collected in the recording phase, and it maps the binary, using the debug symbols, to correlate the jump addresses with functions. Thereby, it associates the performance measurement profile at the granularity of functions. Lastly, TEE-PERF is integrated with Flame Graphs [1] to visualize the performance bottlenecks.

B. Design Details

We next detail the four stages of TEE-PERF.

Stage #1: Compiler. In the compiler phase, we recompile the input application to inject profiling code, which collects the necessary information for the analyzer. Further, it maps code to the binary, and also sets up the communication wrapper for the recorder. While recompiling, we instruct the compiler to inject code at method call and return points. This is supported by `gcc` and `clang`, with function instrumentation and header inclusion in every compile unit before any other include.

The injected code will read a counter value from a suitable source, like the software counter provided by the recorder, and collect the address of the call or return. Further, the injected code sets up a *log*, which is used to collect the performance measurement. The log is set up before any of the measured code is being executed. We can link the setup and tear down code either statically or dynamically.

At a high-level, the log structure consists of the log header and multiple log entries. Figure 2 (a) shows the log header format. The header stores flags, version number of the log structure, the memory address mapping of the shared memory, the process ID of the profiled application, the maximum size of the log structure, an index to the tail of the log for the next log entry write, and a pointer to a well known function entry (address of profiler).

The flags contain for example, if the measurement is currently active, and which events should be measured. These flags are stored in a data structure, which can be atomically read and written by the underlying hardware (HW) platform. This allows to change the flags while the application is executed, without introducing critical sections into the execution, which could become a bottleneck and alter the performance characteristics of the measured application. While the flags can be changed during the execution, and thereby control the recorder, the version number is used to support different

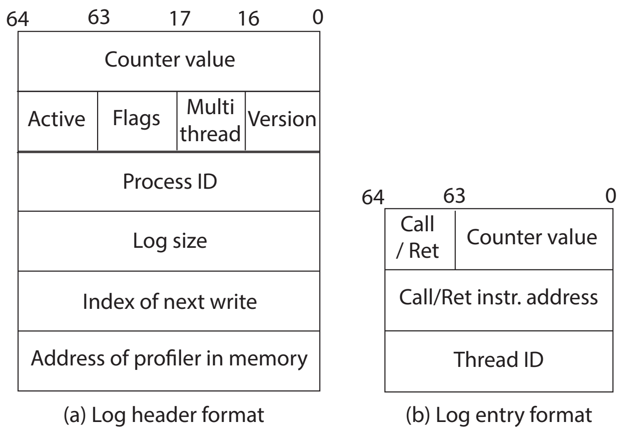

Figure 2: Log format of TEE-PERF

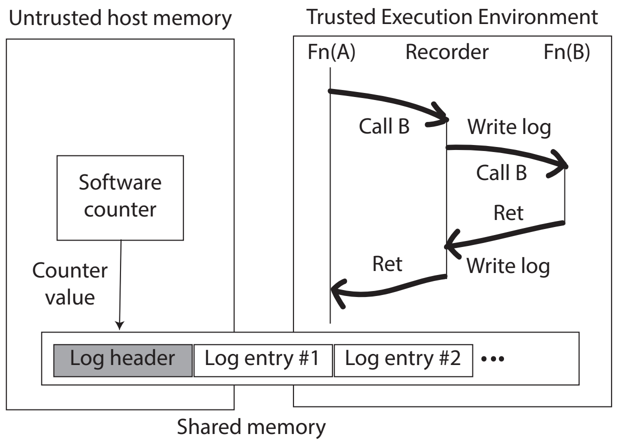

Figure 3: Recorder overview

log structure layouts in the analyzer and is static after it is written once. Therefore, the version number does not have to be accessible atomically.

The process ID is used to differentiate multiple runs or multiple application from each other in the analyzing phase. The maximum size of the log is determined at the beginning, and afterwards it is immutable. It is used in the recorder to know when the log is full, and for the analyzer to dismiss records, which might be wrong at the end of the log.

The pointer to the tail of the log has to be incremented atomically, and it stores the first position of the first empty entry. Any thread which wants to write to the log executes a fetch-and-add instruction on the pointer, guaranteeing that each possible entry is only written once. Since the order of call/returns is only important within the same thread, the possibility of unfair access to the tail does not change the result of the analyzer.

We also added a pointer to a method, which is added by the recorder, to be able to easily determine the mapping offset of relocatable code. This information is necessary to correlate an instruction pointer (IP) with a function in the object and DWARF file.

Each log entry itself consists of 4 data fields (see Figure 2 (b)), storing if the executed instruction was a call or return, the current timestamp, the call or return address, and the threadID of the thread executing the instruction. (We also support multi-threaded applications (see §II-C), which means we also need to record the thread ID as part of the recorded log.)

Stage #2: Recorder. The recorder is the run-time component of TEE-PERF. It consists of two parts (See Figure 3): the code injected in the compiler phase, and the recorder wrapper. The recorder wrapper sets up a shared memory region between the measured application and the wrapper. Since the shared memory is mapped between the TEE and the host, it should not increase the TEE's memory, which is usually limited.

The linked in library maps the shared memory region into the measured applications address space and announces its position through a globally accessible variable. Furthermore, the library will initialize the shared memory to a known state before any of the records can be written.

When a call or return instruction is executed, the program instead of jumping directly to the call/return address will jump to the injected code, i.e. function instrumentation. The instrumented code writes the address of the call/return target, the type of the instruction: call or return, instruction together with the current timestamp, and the threadID into the shared memory (see the log entry in Figure 3). Since the log header allows to increase the tail atomically, the injected code can reserve a log entry in the shared memory before writing the entry itself; thus, the writing process to the log is lock-free. While we designed the log in such a way that it can be used lock-free with atomic instructions, TEE-PERF does not actually rely on the availability of these instruction and can use alternative ways of synchronization. The tracing of call/return address can be dynamically de-/activated by the user at any time. Since TEE-PERF is designed for the development phase, and not production environment, it is acceptable, that the recorder leaks information into the host memory.

After the measurement, the recorder wrapper writes the entire log to the persistent storage, allowing the analyzer to read the log file. Additionally, the recorder process is responsible for making the hardware counters accessible for measurement. If no hardware counters are available, for the TEE measurements, the recorder uses a software counter. This software counter is implemented by a thread incrementing a counter in a tight loop. While this sacrifices an entire core to the counter, it also provides a fine and accurate enough clock to be used for measurements. TEE-PERF does method-level relative profiling, thus perfectly accurate counter are not necessary. Furthermore, since the loop is very small and only accesses the header of the log, the cache footprint is very small, which should minimize the performance impact of the counter incrementation on the measured application.

Stage #3: Analyzer. The analyzer reads the entire log file, and thereafter, it groups the call and return entries together. The grouping is done for each thread independently. Since a thread ID is stored in each log entry, the analyzer is able to determine the run of each thread. We can then use the call and returns to build a call stacks for each method. Furthermore, we have the counter at the method entry and the method exit, using this information the analyzer calculates the time spent in the method. It is also able to subtract the time of the method called by the method and infer the real time spent in the method.

Thereafter, it then adds the time of each function execution together for each method. This information is then presented in a sorted way to the programmer. Additionally, the analyzer provides a rich declarative query interface (§II-C), which allows to do more queries on the collected data, giving the developer the tools to investigate further, e.g. which thread called which method how often.

Stage #4: Visualizer. Lastly, we have integrated the output of the analyzer with Flame Graph [1], a popular visualization tool that allows identification of the most frequent code paths quickly and accurately.

C. Additional Design Features

We next present the additional design features supported by TEE-PERF.

Multithreading support. Our TEE-PERF tool supports performance measurements of multi-threaded applications. To do so, we extended the log file format by writing a threadID to the log file. The analyzer can then reconstruct a progress flow of each thread by sorting the log entries by the threadID. While our tool cannot guarantee that each recorded method enter and exit event is correctly ordered in the log, it can guarantee that for each single thread, as the recorder holds the execution of the thread until the corresponding log entry is written.

Note that the access to the log, while recording, is lock-free, due to the append only nature and the use of atomic instructions. Therefore, we keep the overhead of writing to the log to a minimum.

Call stack. We further support full reconstruction of the call stack. Since the recorder writes every method enter and exit into the log, we are able to fully reconstruct the call stack of every single call. This not only allows us to make accurate timing calculation for every method, but also allows us to support more complicated queries, e.g. performance depending on the call history of a method.

Queries. After the analyzer has read the log, the user can issue further queries. We support a rich query interface for analyzing the log. The interface is based on the declarative Pandas API for python. With these queries, we search for contention in the code or dependencies of calls, which result in a high overhead.

Selective code profiling. We also support selective code profiling. In particular, by selecting parts of the code, where our tool injects the measurements it is possible to only measure parts of the application. Therefore, we provide a systematic knob to reduce the log size and selective code profiling.

III. IMPLEMENTATION

We next present the implementation details of TEE-PERF.

Compiler pass. We use compiler flags which are available in `gcc` and `clang` for injecting profiling code in the application. These flags are `-finstrument-function`, adding a `__cyg_profiler_func_enter` and `__cyg_profiler_func_exit` to function calls or return instruction, respectively. Another feature we use is the `--include` flag, allowing us to include a file in every compilation unit. The included file has the code necessary to write to the log. Therefore, the compiler and/or linker should be able to inline these methods, reducing their overhead further. The final step is to link against the library containing the setup and tear down code. A full compiler call is as follows: `gcc -finstrument-function --include=profiler.h test.c -o test -lprofiler`.

Recorder. The two parts of the recorder are the wrapper and the code injected into the application. Since the recorder should be platform agnostic it does not use any special libraries. Therefore, it only depends on the libc. For compatibility to most environments, it is written in C. Thereby, 389 LoC will be injected into the application, and the wrapper consist of 230 LoC.

Importantly, the injected code has to prevent to be measured itself, as this would result in an infinity loop. We avert that from happening by adding the `__attribute__((no_instrument_function))` to all injected methods.

Analyzer. The analyzer runs offline and potentially on a different system, thus it is not constraint by the same portability considerations as the recorder. Therefore, we implemented the analyzer in Python 3 and used *numpy* and *pandas* for the analysis of the log. Further, to reduce the implementation effort the analyzer depends on UNIX tools: *addr2line*, *readelf* and *c++filt*. We implemented the analyzer with 370 LoC.

In addition, we provide a query interface for analyzing the logs. The query feature was implemented by starting the script in interactive mode. After the script run through the user can issue pandas queries on our data structures to find the information relevant to the user.

Visualizer. We further provide visualization mechanism for the developers to track the performance bottlenecks. Therefore, we provide support for *flamegraph* for visualization. This is directly implemented in the analyzer. Due to the already existing analysis of the log structure, the Flame Graph output could be implemented with as little as 15 LoC. We suspect that other tooling support for visualization should be similarly easy to port.

IV. EVALUATION

Our evaluation answers the following questions:

- What are the profiling overheads of TEE-PERF compared to `perf`? (§IV-B)
- Does TEE-PERF detect performance optimization opportunities for applications running in the TEEs? (§IV-C)

A. Experimental Setup

Experimental testbed. We used a machine with Intel Xeon E3-1270 v5 (3.60 GHz, 4 cores, 8 hyper-threads) with 64 GiB RAM running Linux kernel 4.9. Each core has private 32 KiB L1 and 256 KiB L2 caches, and all cores share a 8 MiB L3 cache. For the storage device our testbed uses a Intel DC P3700 SSD. The SSD has a capacity of 400 GB and is connected over PCIe x4.

Applications and compiler. We evaluated TEE-PERF with applications from the Phoenix 2.0 multithreaded benchmark suite [25]. In addition, we have evaluated TEE-PERF using two real-world applications: *(i)* RocksDB [27] persistent key-value storage: we evaluated RocksDB using the RocksDB benchmark [7]; and *(ii)* Intel SPDK library [18] version for high-performance direct I/O storage operation.

All native applications were compiled using gcc (Debian 6.3.0-18+deb9u1) compiler, and to compile the applications with the profiler we used x86_64-linux-musl-gcc (GCC) 7.3.0. We used compilers with the -O3 optimization flag.

Methodology. We used the Fex [23] framework to run the experiments. For all measurements, we report the geometric mean over 10 runs across all benchmarks.

B. Performance Overheads

We first present the profiling overheads of TEE-PERF compared to `perf`. To measure the overheads, we used the the Phoenix multithreaded benchmark suite running inside the Intel SGX enclave using SCONE [10].

Figure 4 shows that the performance overheads of the TEE-PERF vary significantly across benchmarks. The mean overhead of our tool compared to perf is $1.9\times$. TEE-PERF is in nearly all benchmarks slower than perf as the inject code has to run on each method call and return. In `linear_regression` TEE-PERF is around 8 % faster than `perf`. This is expected as this particular application is issuing a low number of function calls, and is mainly performing computation work inside one function, therefore, the injected code is not executed often. However, `perf` still has to perform context switches to sample the data periodically. The other outlier is `string_match`, where we suffer a $5.7\times$ overhead compared to `perf`. This is due to a high number of function calls, as a result the injected code in our tool is executed often and producing a higher overhead.

We note that the performance debugging is not done in a production environment, but in a development environment. Therefore, even though the performance overheads are on the slightly higher side, they are still quite reasonable considering the useful insights provided for the application programmers.

We also measured the performance of RocksDB with our tool and plotted it using the Flame Graph. Figure 5 shows the runtime of different method with the `db_bench` tool. We run a random read write benchmark with 80 % reads. The Flame Graph shows that the benchmark spent most of its time in getting a current timestamp (`rocksdb::Stats::Now`) and generating random numbers (`rocksdb::RandomGenerator::RandomGenerator`). To increase the performance of RocksDB in the enclave, these two functions either have to be removed from the critical path, or have to be replaced.

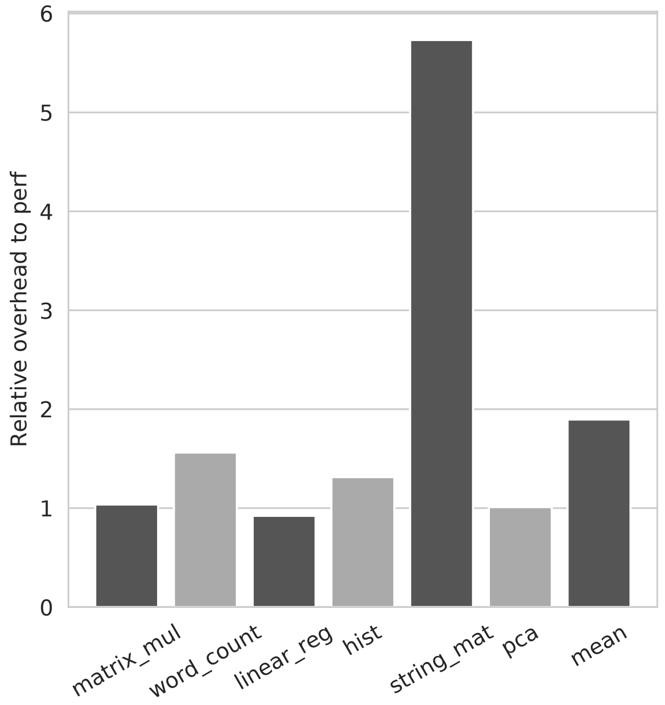

Figure 4: Overhead of TEE-PERF compared to `perf` for the Phoenix benchmark suite running in the Intel SGX TEE.

C. Case-study: Performance Optimization of Intel SPDK

We next present a case-study of Intel SPDK [18], where we used TEE-PERF to optimize the performance in the context of Intel SGX. In particular, entering and exiting a TEE is performance expensive because the hardware needs to perform a secure context switch, e.g. TLB flush. This results in TEE having huge overheads in tasks in which such operations are frequent. One of these operations is I/O access since it requires to make system calls, which cannot be made from within the TEE since it could leak information to the outside. Due to the overhead some applications like storage system struggle to embrace the TEE technology. However, recent development in high performance direct I/O libraries like Intel's DPDK [4] and SPDK [18] presents an opportunity to eliminate most system calls in the critical path, making high-performance storage [19] and network [31] application feasible for TEEs.

To show that TEE-PERF can be used to optimize bigger application, and to show the potential of TEE-PERF, we ported SPDK to Intel SGX. We used the performance benchmark tool which is shipped with SPDK, and measured the performance of SPDK without further optimization inside of SGX. The benchmark we run for measurements was a random read write benchmark with 80 % reads. We found that the naively ported version suffered severely from running

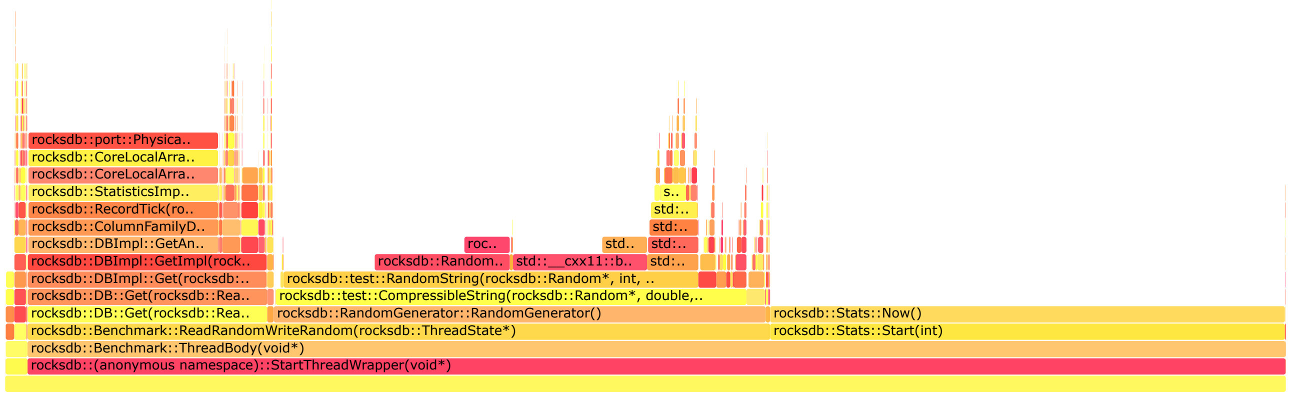

Figure 5: Flame Graph of RocksDB measured by TEE-PERF

inside the enclave. While native SPDK, running on the host system, reached $223,808$ IOPS and a throughput of $874\,\text{MiB}$ for $4\,\text{KiB}$ blocks, the naive unoptimized implementation only accomplished $15,821$ IOPS and a throughput of $61.8\,\text{MiB}$ on the same machine, and $4\,\text{KiB}$ blocks.

Figure 6 (top Flame Graph) shows the Flame Graph for the unoptimized version of SPDK created with TEE-PERF. The Flame Graph shows that the performance tool spends nearly $72\,\%$ of its time in a system call to get the current process ID, i.e. `getpid`. Further, $20\,\%$ are spent in receiving the current time stamp, i.e. `rdtsc`.

After we identified these bottlenecks using TEE-PERF, we implemented a caching algorithm for the `getpid` systemcall, return after the first call the result from the first. While caching of the process ID is unproblematic, the same is not true for the timestamp. We implemented a caching with correcting after a specific amount of calls. This allowed us to reduce the time spent in receiving a timestamp dramatically.

Figure 6 (bottom Flame Graph) shows the optimized call stacks for SPDK plotted using TEE-PERF.

The optimization reduced the time spent in these two methods, while reading and writing to nearly 0. And more time can be spent in reading and writing part of the benchmark.

With the improvements we made the performance of SPDK inside the enclave improved to native performance with $232,736$ IOPS and $909\,\text{MiB}$ per second in the same setup as before. That is an improvement of the factor $14.7\,\times$ compared to the naive unoptimized implementation. Our TEE-PERF tool allowed us to easily detect and identify the bottlenecks of SPDK inside the SGX enclave. After the identification of the bottlenecks, we could easily implement the improvements, which in turn do not suffer from the same performance penalties added by running inside an SGX enclave.

V. RELATED WORK

To our knowledge, TEE-PERF is the first architecture and platform agnostic performance measurement tool for TEEs. We have developed the tool in the context of our Speicher [11] project, a secure LSM-based storage system. As shown also in the evaluation section, Linux `perf` [6] provides similar insights by instrumenting CPU hardware registers but does not offer a detailed log with the time spent on each function. In addition, `perf` is restricted to the Linux environment, and requires the availability of hardware counters. Intel's own VTune Amplifier [5], a proprietary commercial low level analysis tool, offers various features including stack sampling and thread profiling on SGX enclaves. In addition, it provides the time spent on each subroutine down to instruction level but relies on the existence of special HW features and is platform-dependent. Nevertheless, Intel VTune Amplifier is restricted by the target platform (only Intel CPUs). sgx-gdb [17] is debugger extending GDB [2] for enclaves. It allows stack sampling, which makes it possible to profile applications running in an SGX enclave. However, sgx-gdb is restricted to Intel SGX and Linux.

SGX-Perf [34] is a recently proposed profiler for Intel SGX. It profiles the enclave enter and exit events. The primary function of SGX-perf is to analyze the cost of context switches in the SGX enclaves. Unfortunately, SGX-Perf does not provide method-level profiling, as supported by TEE-PERF. Further, SGX-Perf specifically designed for Intel SGX, whereas TEE-PERF is not limited to the Intel SGX architecture.

Another tool which works similarly to perf is Gprof [3], an extension of the original Unix prof, which offers performance analysis for Unix applications by sampling and instrumenting code at compile time. It offers a flat profile of total execution time, broken down by function, and a call graph, which shows function invocations. However, contrarily to TEE-PERF, it does not provide cross-platform support. Further, Flamegraph allows better comprehension compared to call graphs.

LIKWID [32] proposed a low-level lightweight profiling suite for the x86 architecture. However, it is tightly coupled with the x86 architecture since it relies on the hardware counters to provide the performance profiling metrics.

MemProf [21] and Memphis [22] provide profiling information for the remote memory accesses of memory objects on a NUMA architecture. These profilers are very important to perform NUMA-specific optimizations for remote memory

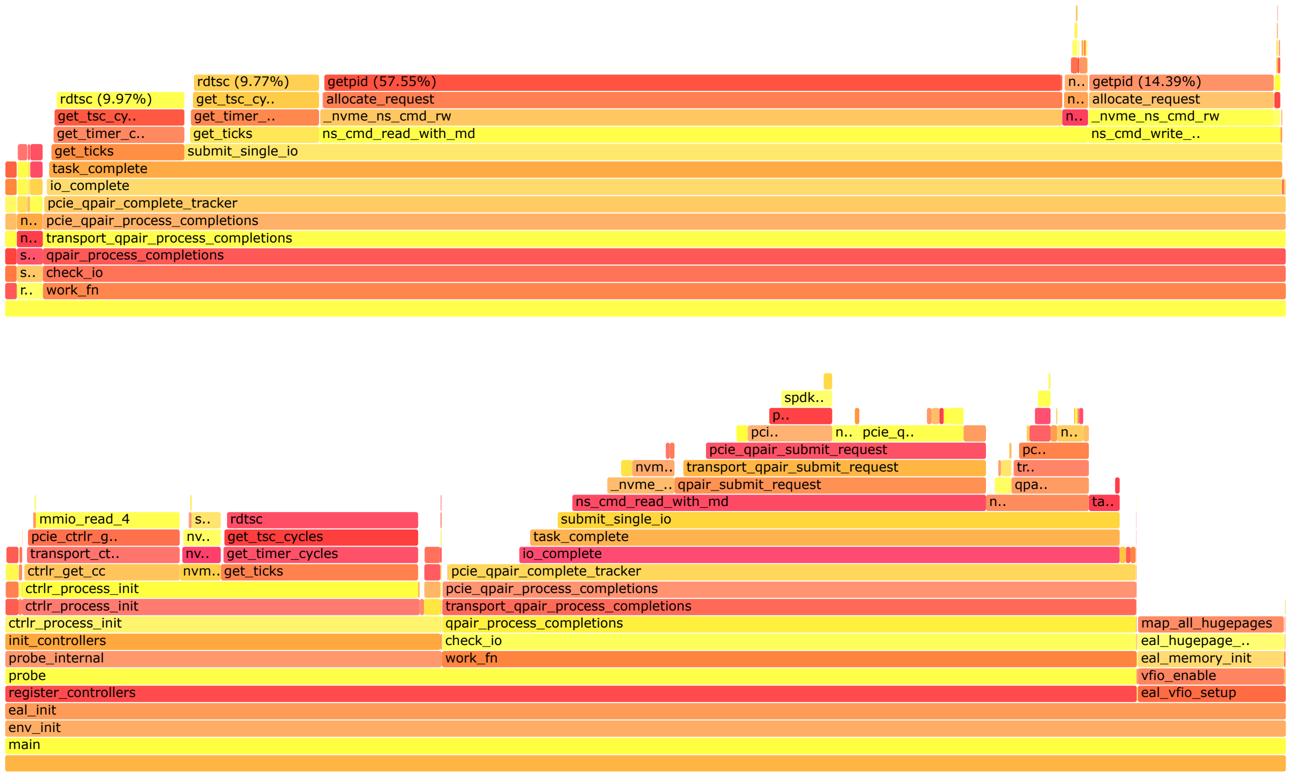

Figure 6: Flame Graphs for Intel SPDK running inside Intel SGX enclave plotted using TEE-PERF: **(top)** unoptimized version of SPDK, and **(bottom)** optimized version of SPDK.

accesses, e.g. thread or page pinning to minimize the remote memory accesses. However, both profilers rely on hardware-specific counters, such as instruction-based sampling (IBS). Further, they are tightly integrated with Linux `perf` to identify memory accesses. In contrast to MemProf and Memphis, TEE-PERF does not require any architecture- and platform-specific counters. Furthermore, our profiler is geared for the trusted execution environments.

Coz [14] introduced causal profiling to locate optimization opportunities in concurrent applications. The key idea in Coz is to slowdown the execution to have the same relative effect of virtually speeding up the code section. In this way, it is able to identify causal relationship between two code segments executing concurrently. Likewise, Coz, TEE-PERF targets unmodified multi-threaded applications. In contrast to Coz, TEE-PERF targets profiling applications running inside the TEEs. More importantly, Coz also relies on Linux `perf` to collect the program counter and user-space call stack. In contrast, our approach is completely platform independent.

Inspector [29] proposed a data provenance library for unmodified multithreaded applications to provide detailed information about thread schedules, and memory accesses. However, Inspector relies on Linux `perf` and Intel Processor Trace (Intel PT) to provide these performance statistics. Our approach does not require Intel PT to trace the execution.

Profilers for distributed systems, such as Conductor [35], Fay [15], and Sieve [30], target the same research direction. These systems provide detailed overview of performance bottlenecks in the applications. However, these profilers target distributed systems, and they are orthogonal to our work since we are currently targeting single-node systems. Moreover, our primary focus is on applications running inside the TEEs.

VI. CONCLUSION

In this paper, we presented TEE-PERF, an architecture- and platform-independent profiler for TEEs. TEE-PERF supports unmodified multithreaded applications without relying on TEE-specific hardware counters or platform-specific kernel features. Further, TEE-PERF supports accurate method-level profiling without employing instruction pointer sampling. We have implemented TEE-PERF with an easy to use interface, and integrated it with Flame Graphs to visualize performance bottlenecks. We evaluated TEE-PERF based on a multithreaded benchmark suite and real-world applications. Our experimental evaluation shows that TEE-PERF incurs low profiling overheads, while providing accurate profile measurements compared to `perf`.

Software availability. TEE-PERF along with the entire experimental is available: https://github.com/mbailleu/tee-perf

Acknowledgements. We thank anonymous reviewers and our shepherd Sara Bouchenak for their helpful comments. This work was supported by the European Unions Horizon 2020 research and innovation program under grant agreement 645011 (SERECA), Huawei Research, and the UK Research Institute in Secure Hardware and Embedded Systems (RISE).

References

[1] Flame Graphs. http://www.brendangregg.com/flamegraphs.html.

[2] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[3] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[4] Intel DPDK. http://dpdk.org/.

[5] Intel VTune Amplifier. https://software.intel.com/en-us/vtune.

[6] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[7] RocksDB Benchmarking Tool. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[8] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[9] ARM. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[10] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[11] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[12] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] V. Costan and S. Devadas. Intel SGX Explained, 2016.

[14] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[15] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[16] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen. Performance of trusted computing in cloud infrastructures with intel sgx. In *International Conference on Cloud Computing and Services Science (CLOSER)*, 2017.

[17] Intel Software Guard Extensions SDK for Linux OS. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf. Last accessed: Dec, 2018.

[18] Intel Storage Performance Development Kit. http://www.spdk.io. Last accessed: Dec, 2018.

[19] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[20] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[21] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the USENIX Conference on Annual Technical Conference (USENIX ATC)*, 2012.

[22] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[23] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. Fex: A software systems evaluator. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[24] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*, 2017.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[26] RISC-V. Keystone Open-source Secure Hardware Enclave. https://keystone-enclave.org/.

[27] RocksDB — A persistent key-value store. https://rocksdb.org/.

[28] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[29] J. Thalheim, P. Bhatotia, and C. Fetzer. INSPECTOR: Data Provenance Using Intel Processor Trace (PT). In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[30] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware)*, 2017.

[31] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[32] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW)*, 2010.

[33] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[34] N. Weichbrodt, P.-L. Aublin, and R. Kapitza. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2018.

[35] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.