Secure Identification of Actively Executed Code on a Generic Trusted Component

Bruno Vavala[1,2], Nuno Neves[2], Peter Steenkiste[1]
[1]CSD, Carnegie Mellon University, U.S.
[2]LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

***Abstract*—Code identity is a fundamental concept for authenticated operations in Trusted Computing. In today's approach, the overhead of assigning an identity to a protected service increases linearly with the service code size. In addition, service code size continues to grow to accommodate richer services. This trend negatively impacts either the security or the efficiency of current protocols for trusted executions.**

We present an execution protocol that breaks the dependency between the code size of the service and the identification overhead, without affecting security, and that works on different trusted components. This is achieved by computing an identity for each of the code modules that are actually executed, and then building a robust chain of trust that links them together for efficient verification. We implemented and applied our protocol to a widely-deployed database engine, improving query-processing time up to 2× compared to the monolithic execution of the engine.

I. Introduction

The continuous growth of Cloud Computing today has increasingly fueled research on new security techniques. In the past decade, special focus has been given to the protection of security-sensitive application modules running on untrusted third-party (UTP) platforms through the use of Trusted Computing Components (TCCs). In the research literature, several trusted execution architectures have been proposed [13, 42, 33, 21, 32, 8, 23, 31]. In the industry, on the other hand, TCCs are primarily used for storing cryptographic material, e.g., as in BitLocker [35] and Amazon CloudHSM [6], but prototypes are emerging for securing complex software [25, 10].

In the Trusted Computing area, code identification [17] is a key mechanism for guaranteeing execution integrity. It consists of: computing and attesting the identity of some code `c` on the UTP side; and then verifying both the attestation and the code identity on the client side. More precisely, the UTP includes a TCC that computes `c`'s identity by hashing it. The TCC then digitally signs `c`'s identity and sends the signed identity to the client. This allows the client to verify that the correct code was executed. By extending the same procedure to the input and output data, the client is also able to verify that the received output was obtained by running `c` with the correct input.

The major challenge with using code identification for securing increasingly complex software is that the overhead to compute `c`'s identity before the execution grows linearly with `c`'s size. In particular, the overhead scales with the size of the code that *may* be executed, not the size of the code modules that *are* actually executed. Consequently, this becomes a concern when the actively executed code is only a fraction of the code base. In addition, such overhead should not be considered as a one-time burden. In fact, frequent code identification is desirable to refresh the execution integrity property, which is otherwise guaranteed only at load time.

In this paper we present a protocol for code identification and execution that breaks the coupling between code size and cost of identification. The protocol has two key desirable features. First, the trusted architecture loads, identifies and runs only modules of the code base that are actually executed. This provides execution flexibility to the UTP and saves TCC resources. Second, the correct execution sequence of code modules is guaranteed by a robust and verifiable execution chain. Each module secures the application data using a secret key that depends on its own identity and the identity of the next module in the correct sequence. These two mechanisms combined enable a secure and efficient identification.

The protocol further enables efficient verification on the client side. In fact, the client checks that the correct code was executed by only verifying a chain endpoint to bootstrap trust in the whole chain. The client does not need to be aware a priori of the exact execution order; also, unused code modules have to be neither loaded nor verified.

Additionally, our protocol is agnostic to the details of the TCC, which makes our contribution generally applicable. The protocol performs downcalls (to the TCC) by means of a simple and generic interface. Five primitives (including one on the client side for execution verification) represent the bridge between the protocol and the trusted computing services provided by the TCC, such as isolated code execution, attestation and secure storage. These services are available, or implementable, on different trusted components.

We make the following contributions:

- We design an efficient protocol for the secure execution of complex software inside a generic trusted component. The cost of code identification scales with the size of the modules that are executed, rather than with the size of the service code base.
- We analyze the security of our constructions. Also, we introduce a novel zero-round key sharing technique for trusted executions that improves performance with minimal changes.
- We implement the protocol on a hypervisor-based TCC (XMHF/TrustVisor [32, 46]). We apply it to the widely-deployed SQLite DB engine [40], formally verify its correctness, and show its performance benefits.

The paper is organized as follows. We elaborate on the challenges of securing remote code execution and outline our solution in Section II. We provide background on TCCs in Section III and define our protocol in Sections IV. We describe our implementation, formal verification and experimental eval-

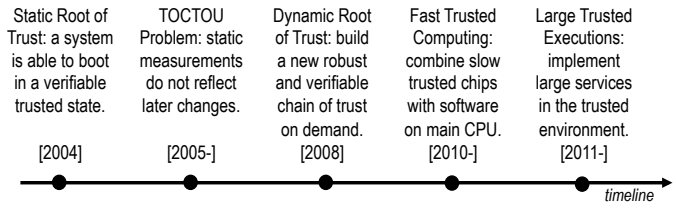

Fig. 1: Trends in Trusted Computing research work show an initial focus on reducing the Trusted Computing Base (TCB), while recent advances in technology enable to secure entire unmodified services, thus enlarging the TCB.

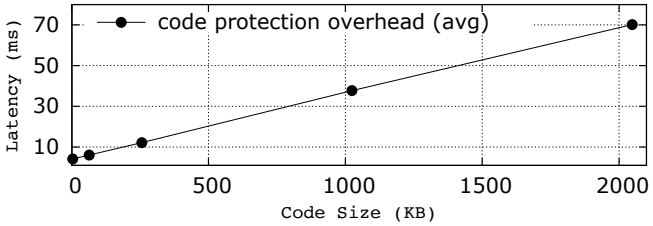

Fig. 2: Security-sensitive code registration latency in XMHF/TrustVisor. It shows a linear dependence between code size and protection overhead.

uation in Section V. In Section VI we devise and validate a performance model for code identification. We review related work and draw conclusions in Section VII and VIII.

II. Towards Flexible Trusted Executions

Current trends (Fig. 1) in Trusted Computing (TC) evidence that the code used in trusted executions is growing. We show that this raises either efficiency or security concerns in TC architectures, and that this is a result of how code identification is done today [17]. This helps us defining the problem statement, goals, and outlining our solution.

A. Previous Work

Early work used trusted hardware to verify the integrity of a system's initial state [37, 26]. The mechanism involves identifying—taking integrity measurements, i.e., hashing—the software components (e.g., BIOS, boot loader, OS, applications) that bring the system into an operative state. The identities are stored on trusted hardware (e.g., a TPM) and conveyed to a client through an attestation. The client bootstraps trust in the system's initial state by verifying the validity of the attestation and matching the identities with the expected ones.

Preserving trust during the execution is hard. Operating systems are constantly subject to attacks; vulnerabilities are discovered on a daily basis [2]; and tools are available to exploit them [1]. Hence, the guarantee that a system is trusted at a certain point in time may not hold later—this is also known as time-of-check-time-of-use (TOCTOU) gap [38, 11].

This gap was reduced through the notion of *late launch* [20] to create a Measured Launch Environment on demand. Flicker [33] shows that the technology can be used to run a security-sensitive piece of code in isolation. The result is a dramatic reduction in TCB size and, consequently, of the attack surface. Subsequently, given the poor performance of the low-power hardware modules, solutions were devised to speed up the computation [32, 8]. Instead of relying directly on the hardware module, trust is extended to a small trusted software module that enables faster code identification. Recently, faster trusted hardware has also been proposed [34].

Improvements on TC technology have made it possible to grow the code base from a few KB to hundreds of MB. In fact, in order to provide security guarantees to a broader set of applications, some works have secured entire database engines [9, 47], and even unmodified Windows binaries such as SQL Server and Apache HTTP Server [10].

B. Security or Efficiency, But Not Both

There are currently two alternatives to deal with such large code bases, and both come with big downsides. We dub the first as *measure-once-execute-forever* [10]. The integrity measurement is taken only before the execution of the code, which then continues in the trusted environment indefinitely. This approach brings us back to the TOCTOU problem. Since the integrity measurement of a code base is only taken once, it will not detect any later successful attack that compromises it.

The second alternative is instead dubbed *measure-once-execute-once*. The measurements are repeated before each execution (e.g., a Flicker [33] based application). This approach instead may raise efficiency issues. In fact, in order to assign an identity to the code, this must be loaded first and then hashed.

As an example, in Fig. 2 we quantify this load-and-hash cost on XMHF/TrustVisor [32, 46], a recent tool for efficient and secure code execution. We measured the time to register different code bases. During this procedure, the memory pages of the code are isolated and identified. The time scales linearly with the code size reaching about $37ms$ for just 1MB of code.

Such a linear dependence holds also for Intel SGX [25, 34], used to build secure Enclaves. In fact, after an Enclave is created and protected (`ENCLS[ECREATE]` instruction), code pages must be added and measured (`ENCLS[EADD]`, `ENCLS[EEXTEND]`), before finalizing the Enclave and fix its identity (`ENCLS[EINIT]`). Hence, the overhead of creating an Enclave identity grows with the code size. We lack however an SGX-enabled platform to measure the (likely lower) slope.

C. Problem Definition

Clearly, the code identification cost has become a bottleneck. If the code is identified only *once*, identity integrity stales over time; if the code is identified *repeatedly*, the overall execution time may increase considerably for large code bases. The ideal balance is to have non-stale identities and an execution time less dependent from code base size.

In this paper, we aim at making the secure execution cost scale with the size of the actually executed code, instead of the size of the code base as a whole, independently from the used trusted component. Such generally-applicable method can balance the cost of re-identifying some code to refresh integrity guarantees and further reduce the active TCB size.

In summary, we seek to attain the following properties:

1) **Secure proof of execution**. The proof of execution of the correct code must be unforgeable, unambiguous, and linked to the hardware root of trust.
2) **Low TCC resource usage**. The protocol should achieve security with minimal resource (code identification, cryptography, storage, etc.) demand on the TCC.
3) **Verification efficiency**. The overhead for the client should be constant, independently from the code base—i.e., a fixed number of hashes and digital signatures.
4) **Communication efficiency**. The protocol should be "non-

interactive", requiring only a small additional constant amount of traffic to enable successful client verification.

5) **TCC agnostic execution**. The protocol should use any underlying TC architecture as a black-box, thus allowing to retrofit existing trusted components.

D. Overview of our Solution

The core of our solution is displayed in Fig. 3. On the left, the code base is depicted as a set of logically connected modules (arrows express the control flow graph) stored on the UTP, working together to provide a service. On the right, our protocol works as follows. It executes in the sequence on the TCC ① the modules of the code base that are necessary for the requested service (e.g., module *C* is not loaded). Given a particular client request, only the modules required to serve it are considered active (*A* and *B* in the figure). Active modules are loaded and run according to the correct execution order. Each module secures ② the intermediate state before it terminates. The next active module is then executed ③ and it validates the previous intermediate state ④. Such state is passed through modules by means of logical secure channels.

In contrast to having a single identity assigned to the code base, each module has its own identity. This identity is calculated as the hash of the code. This allows us to maintain backward compatibility, so to achieve a general solution that is implementable on current TCCs. Our protocol builds a robust execution chain based on the identities of the modules, and guarantees that the modules are executed in the correct order with respect to the control flow graph.

Each executing active module has access to data and resources required for the computation. Before the execution, each module is expected to receive some input from the client (e.g., a request) or some intermediate state from other modules. Similarly, when it terminates, each module is expected to produce either an output for the client (e.g., a reply) or some intermediate state to be processed by the next module in the execution flow. Before and after the execution, every piece of data to/from any module is handled by the UTP in the untrusted environment; consequently, it must be secured by means of the available TCC resources (e.g., secure storage).

An executing active module has access to the Identity Table (or identity set) of the code base. Such module can thus leverage TCC-based access control mechanisms to secure data. Intermediate states are transferred between modules through logical identity-dependent secure channels, whose security is enforced by the TCC. Such channels are "logical" in that the data is transferred through modules by the UTP (i.e., *A* releases data to the UTP and it is unloaded, then *B* is loaded and receives data from the UTP). The channel is secure as it guarantees (1) data integrity, through message authentication codes, and (2) the authentication of the end points based on their execution order (i.e., *A* sends to *B* and *B* receives from *A*, but *B* and *C* will never exchange data). A benefit of these channels is that they allow to maintain all the data locally (UTP side) thereby avoiding the interaction with the client during the execution.

A client reply is authenticated through a TCC attestation, or through previously established symmetric secret keys. The client receives and verifies the attestation. The last executed module (in the control flow graph) calls the TCC attestation service, and the TCC attests the module's identity. Such last module includes (the integrity measurement of) some parameters in its attestation, such as the client's initial request, the identity set of the code modules, and the reply. The attestation, jointly with the parameters used to generate it, represents a proof of execution verifiable by the client. By verifying the module's identity and the identity set, the client can trust that the code base correctly served the request.

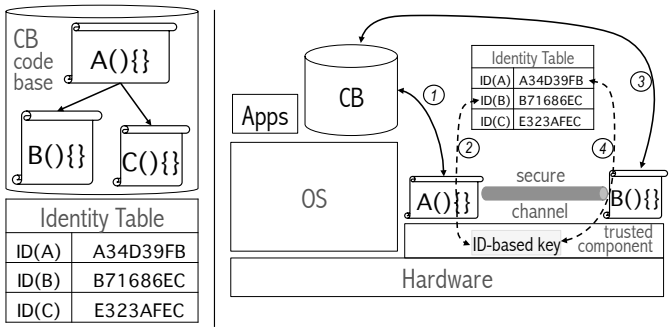

Fig. 3: Sketch of our solution.

Note that, by design, the hash chain created by the protocol enforces the execution order of the modules and guarantees their integrity by computing their identities. This means that the client neither has to be aware of the execution order for any specific execution, a highly desirable feature, nor has to verify the identity of any modules, except for the last one.

III. Model

Threat Model. The UTP platform is untrusted, though equipped with a trusted component. An adversary may take control of *any* software running on the UTP machine, including services and the OS. So, he can read/modify any data.

The UTP platform is equipped with a TCC. The TCC is responsible to provide security guarantees for code and data, and to link them to the (trusted) TCC manufacturer. The TCC is trustworthy due to its minimal hardware/software security perimeter (i.e., it does not include peripherals, such as disk or network devices, nor the OS). A TPM-based TCC includes components such as: the CPU, the LPC bus, the memory modules, the communication bus with the memory. Other implementations could be based on secure coprocessors [39].

The adversary cannot perform physical attacks but is allowed to use the TCC. Actually, we assume that the TCC primitives (described below) are always called by an untrusted principal. Hence, the adversary can tamper with the intermediate states of code modules when they are not running in the trusted environment. It can inject false data as input to a code module and it can execute tampered modules on the TCC.

DoS and cryptographic attacks are out of scope. The former are difficult to prevent due to the untrusted OS. The latter are assumed computationally infeasible for the adversary.

System Model. *Code base.* Our service is composed by m modules (or PALs[1]) $\mathtt{p}_1, \ldots, \mathtt{p}_m$. The control flow is a directed graph over the PALs describing their execution order. An execution flow is a sequence of PALs of finite but unknown length $\mathtt{n}$ (e.g., $\mathtt{p}_1, \mathtt{p}_3, \ldots, \mathtt{p}_4$) that respects the control flow. We

[1]Piece of Application Logic, using the notation of previous works [33, 32].

refer to a generic execution flow with $\mathtt{p}_1, \ldots, \mathtt{p}_n$. It will be clear from context the distinction with the code base set.

We assume the service is either originally created or suitably partitioned into code modules by the service authors. In Section VII, we briefly discuss how modules can be defined.

UTP-*side.* The code base is available on the UTP, possibly outsourced by its (trusted) authors.

Client-side. The client knows the cryptographic hashes of the attested PALs, and also the hash of the Identity Table (Fig. 3), which represents the identity set. Ideally, the information could be provided by the (trusted) authors of the code base and it requires a constant amount of space.

Additionally, the client knows and trusts the TCC's public key $\mathtt{K}^+_{\mathrm{TCC}}$. This can be achieved through an initial TCC Verification Phase: the client interacts with the UTP to retrieve $\mathtt{K}^+_{\mathrm{TCC}}$ and the associated certificate. If the public key is correctly certified by a trusted Certification Authority (e.g., the TCC manufacturer), then it can be trusted and used for verification.

TCC Model. The TCC is abstracted through a set of primitives, derived from the available Trusted Computing technologies [20, 45]. They can be implemented on a variety of TC architectures, e.g., Intel TXT-enabled processors and a TPM. Future implementations may leverage Intel SGX [25].

- $\mathtt{out} \leftarrow \mathsf{execute}(\mathtt{c}, \mathtt{in})$ It executes some code `c` in isolation over some input data `in` and returns the output `out`.
- $\{\mathtt{data}\}_{\mathtt{K}}^{\mathtt{snd-rcv}} \leftarrow \mathsf{auth\_put}(\mathtt{rcv}, \mathtt{data})$ and $\{\mathtt{data}, \emptyset\} \leftarrow \mathsf{auth\_get}(\mathtt{snd}, \{\mathtt{data}\}_{\mathtt{K}}^{\mathtt{snd-rcv}})$ The primitives for secure storage allow to specify the identity `rcv` (resp. `snd`) of the only recipient (resp. sender) code that can retrieve (resp. send) some `data`. Secured data is protected with key `K` and stored outside the TCC in untrusted storage.
- $\mathtt{report} \leftarrow \mathsf{attest}(\mathtt{N}, \mathtt{parameters})$ It accepts a fresh nonce `N` and some `parameters`, typically measurements, and produces the attestation (`report`). The attestation binds this information together with the identity of the executing code, which is stored in an internal register REG.
- $\{0, 1\} \leftarrow \mathsf{verify}(\mathtt{c}, \mathtt{parameters}, \mathtt{N}, \mathtt{K}^+_{\mathrm{TCC}}, \mathtt{report})$ It is implemented on the client, who calls it with the execution information (such as code identity, input and output parameters, nonce, TCC public key and report) to be verified.

IV. Secure Identification of Actively Executed Code

A. A Naive Solution

A client could verify and establish trust in the execution of a large code base by iteratively checking that each PAL is run correctly and respects the control flow graph. A relatively simple protocol to achieve this is the following: the client sends a request to the UTP to execute the first PAL $\mathtt{p}_1$ on the TCC, and provides the necessary input values for the service. When the PAL terminates, the UTP forwards to the client an attestation returned by $\mathtt{p}_1$ that covers its identity (i.e., a hash of the module), the input and the output data. The output includes the identity of the next PAL to be run, besides the result of the execution (i.e., the intermediate state). Using the TCC primitives, the client can verify that the output is valid, since it was calculated with the correct code and the proper input. The same procedure can then be repeated for each PAL in the execution flow until the final result (i.e., the actual reply for the client) is produced by $\mathtt{p}_n$. The protocol ensures that the PALs are called in the right order and run over the correct data. Hence, it offers the required correctness guarantees.

Although the naive approach is secure and only attests the code modules that are actually executed, it has a number of drawbacks. First, attestations are expensive, so a large number of executed modules can consume lots of TCC resources. Second, it is interactive, since it requires the client to verify each PAL and to mediate the transfer of the intermediate state between two module executions. Third, it is not verification efficient as the client has to check every attestation.

In the rest of the section we eliminate the above drawbacks. We explain a set of orthogonal techniques that: remove the interactivity with the client and reduce the TCC attestations to one (§IV-B), address an issue with identity-based secure storage (§IV-C), and optimize performance with a novel TCC-based key sharing solution (§IV-D). Finally, we devised a flexible and verifiable trusted execution protocol (§IV-E).

B. Reducing Communication

When a trusted execution is requested, the client is only interested in obtaining the final reply (generated by the last executed module $\mathtt{p}_n$) and in verifying the validity of the whole execution (i.e., as if the code were executed as one single module). As a consequence, the intermediate states do not have to be transmitted to the client as long as the client can check at a later time that they were handled correctly. Similarly, each PAL execution does not need to be attested as long as the client is still able to verify the correctness of the final result.

In the naive protocol, the client is involved in each PAL execution to ensure that the result of a piece of code $\mathtt{p}_i$ is properly provided as input to the next module $\mathtt{p}_{i+1}$. This is accomplished with two attestations returned to the client. The first is generated by $\mathtt{p}_i$ and provides evidence about $\mathtt{p}_i$'s intermediate state and the identity of the PAL that should be run next. The second is generated by $\mathtt{p}_{i+1}$ and provides evidence that the PAL received the correct intermediate state. Therefore, if a malicious UTP tampers with the execution, e.g., by running $\mathtt{p}_{i+1}$ with some incorrect input data, this can be detected with the second attestation. Hence, the verification of these attestations confirms that the intermediate state was correctly transferred from $\mathtt{p}_i$ to $\mathtt{p}_{i+1}$.

Attestations are a key mechanism in secure code execution but the overhead they impose is a concern. Attestations are essential because they convey the execution integrity properties to a client. They are however expensive, since they involve digital signatures. In addition, they are meant to be verified. Consequently, each one imposes a non-zero overhead on the client. Verification requires not only the signature check, but also access to a copy of all data that is attested (i.e., at least the measurement of the code, the input and the output data).

In our approach, we build a "secure channel" between PALs without the client's supervision, thus saving attestation, communication and verification effort. We leverage the TCC secure storage capabilities (§III) to protect the data while it is saved locally in the UTP's untrusted storage. Recall that the TCC secure storage is based on code identity to authenticate the auth_put and auth_get operations. By ensuring that only the correct code modules access security-sensitive intermediate

state results, secure storage can be used as the basis to build a secure data transmission between PALs, instead of relying on attestations. Essentially, a *mutually-authenticated* channel is created: a PAL $\mathtt{p_i}$ authenticates the identity of the previous sender $\mathtt{p_{i-1}}$ when it gets the data from a protected input, and uses the identity of the next recipient $\mathtt{p_{i+1}}$ to securely store its results, before releasing them to the UTP's untrusted environment. It is because of this construction that the client only needs to verify the last executed PAL. Consequently, it is critical to ensure that such single verification can indeed be utilized to bootstrap trust into an arbitrary number of correct (though unverified) PALs that were called previously. We now present the end-to-end scenario for completeness and clarity.

The client first issues a service request and provides the respective input values `in` and a fresh nonce `N` to the UTP. The UTP calls the first module with the input: $\mathsf{execute}(\mathtt{p_1}, \mathtt{in}||\mathtt{N})$. The module carries out the initial part of the service computation and it invokes $\mathsf{auth\_put}(\mathtt{p_2}, \mathtt{h(in)}||\mathtt{N}||\mathtt{out})$ before terminating. In other words, it saves a measurement of the input and any output intermediate state in secure storage, specifying the identity of the *only* subsequent PAL that is allowed to retrieve it (i.e., $\mathtt{p_2}$ in the service execution flow). The outcome of the call is the protected data $\{\mathtt{h(in)}||\mathtt{N}||\mathtt{out}\}_K^{\mathtt{p_1}-\mathtt{p_2}}$, which is then returned to the UTP. Notice that $\mathtt{p_1}$ is not attested, so it will not be verified by the client.

The UTP next calls $\mathsf{execute}(\mathtt{p_2}, \{\mathtt{h(in)}||\mathtt{N}||\mathtt{out}\}_K^{\mathtt{p_1}-\mathtt{p_2}})$ to run module $\mathtt{p_2}$. The PAL authenticates the received data to make sure that it came from trusted source. This is achieved by calling $\mathsf{auth\_get}(\mathtt{p_1}, \{\mathtt{h(in)}||\mathtt{N}||\mathtt{out}\}_K^{\mathtt{p_1}-\mathtt{p_2}})$ with $\mathtt{p_1}$'s identity. If the identity is not correct then `auth_get` fails, otherwise it succeeds and the PAL continues (part of) the service execution. Before it terminates, the PAL performs $\mathsf{auth\_put}(\mathtt{p_3}, \mathtt{h(in)}||\mathtt{N}||\mathtt{out})$ to secure the (updated) output intermediate state for the subsequent PAL. This procedure is repeated by all intermediate PALs.

The last PAL is attested and verified by the client. After $\mathtt{p_n}$ retrieves the result from $\mathtt{p_{n-1}}$ and runs the service code, it calls the attestation primitive $\mathsf{attest}(\mathtt{N}, \mathtt{h(in)}||\mathtt{h(out)})$ to get a proof of execution that covers the input and output measurements, besides $\mathtt{p_n}$'s own code. Since the attestation includes the nonce `N`, it also gives assurance about the freshness of the computation. The output with the attestation and the reply data $\{\mathtt{report}, \mathtt{out}\}$ is first released to the UTP's untrusted environment, and then forwarded to the client. The client verifies the execution proof by calling $\mathsf{verify}(\mathtt{p_n}, \mathtt{h(in)}||\mathtt{h(out)}, \mathtt{N}, \mathtt{K}^+_{\mathtt{TCC}}, \mathtt{report})$ and accepts the result only if the primitive succeeds.

Analysis. The attestation binds together the initial inputs, the output and the identity of the last PAL. The cryptographic mutually authenticated chain that links $\mathtt{p_n}$ to the previous PALs ensures that computation is performed *only among correct PALs*: when the verification of the correct execution of $\mathtt{p_n}$ succeeds then, by construction, the client also trusts that $\mathtt{p_n}$ can only have received data from a valid $\mathtt{p_{n-1}}$; the same reasoning can be repeated up to $\mathtt{p_1}$, which is the single entry point to the service and is the PAL that received the initial input data. Hence, correct intermediate PALs only accept data from (respectively deliver data to) correct PALs. Furthermore, as each piece of code specifies the receiver in auth_put, the overall execution order must match a valid control flow.

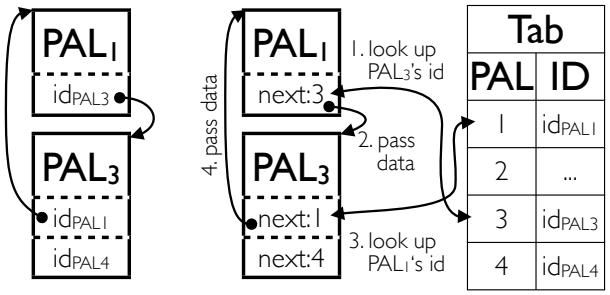

Fig. 4: The *looping PALs problem* (left-side) and our solution of detaching PALs from identities (right-side).

The only data that is accepted inside the trusted environment without being initially validated is the client's input. Similarly, the only data that is released outside without being protected in secure storage is the final output. However, their measurements are included in the last attestation, which allows the verification of the overall execution chain.

Freshness is guaranteed by the client provided nonce, which is propagated throughout the full execution. This prevents attacks where a malicious UTP would replace the output of a PAL $\mathtt{p_i}$ with a value returned by the same PAL in a previous run of the protocol. Notice however that this attack could only succeed if the initial client input values (and so $\mathtt{h(in)}$) were the same in both service executions.

C. Addressing Looping PALs

PALs that exchange their intermediate state through TCC-based secure storage must have access to each others' identities. In auth_put, a module must specify the identity of the next PAL that should be granted access to secured data. Similarly, in auth_get, a PAL must give the identity of the sender module from which it is supposed to receive the data.

A straightforward approach is to include the identities of the next PALs statically in the code of a PAL. Unfortunately, this solution does not work out due to possible loops in the control flow graph of the service that end up creating unsolvable hash loops. Consider the example in Fig. 4 (left-side), where a PAL's code `c` has the identities of other PALs appended to it (e.g., module $\mathtt{p_1}$'s code $\mathtt{c_1}$ contains the identity of PAL $\mathtt{p_3}$, namely $\mathtt{h(p_3)}$). It follows that:

$$\begin{aligned} \mathtt{p_1} &= \mathtt{c_1}||\mathtt{h(p_3)} && = c_1||\mathtt{h}(\mathtt{c_3}||\mathtt{h(p_1)}||\mathtt{h(p_4)}) \\ \mathtt{p_3} &= \mathtt{c_3}||\mathtt{h(p_1)}||\mathtt{h(p_4)} \end{aligned}$$

This example shows that loops in a control flow graph require a module to depend on a hash of itself. Solving the above equations cannot be done for cryptographic hashes as it would require us to violate the properties of these functions. Trapdoor functions could be used instead but they bring a few drawbacks. First, they are typically based on asymmetric cryptography. Hence, they can be comparatively more expensive and further introduce the difficulty of protecting the private key. Second, they do not answer the more fundamental question of whether these hash loops can be avoided. In the following we show how to solve this issue without trapdoor functions.

Our approach uses a *level of indirection* to separate a PAL's code from its identity. We replace the critical identity information, hard-coded inside a PAL, with a lookup operation in a table Tab. The table contains the set of all PALs' identities and is built when the modules are originally created. In addition, we hard-code inside each PAL the index(es) in Tab

$$K_{sndr-rcpt} = \begin{cases} f(K, \text{REG}, rcpt) & \text{on kget\_sndr by } sndr \\ f(K, sndr, \text{REG}) & \text{on kget\_rcpt by } rcpt \end{cases}$$

Fig. 5: Identity-depend key sharing construction for Secure Storage. `rcpt` is the identity of the recipient PAL and `sndr` is the identity of the sending PAL. REG is the register inside the TCC that stores the identity of the currently executing PAL. It is equivalent to a PCR on TPMs [20] or to the $MRENCLAVE$ register in Intel SGX [34]. $f()$ is a keyed hash function.

of the correct next PAL(s) in the control flow (Fig. 4 right-side). The identities thus become independent from each other and each PAL's hash can be computed despite any loop in the control flow graph. The chain that binds our PALs together is now based on Tab, which translates an index into an identity.

Tab is critical to ensure the correct execution flow of the PALs. Hence, it has to be protected throughout the computation of all modules and eventually verified as follows. The first PAL accepts the table as input and propagates it to subsequent PALs using the TCC-based secure storage. The attestation of one PAL—the last executed in the control flow—has to cover the measurement of Tab. In order to eventually verify the execution, the client needs to be aware of: the last executed PAL's identity and the integrity measurement of Tab. Notice that this imposes only a small additional constant space and time overhead for any trusted execution.

The service developers should produce Tab together with the executable modules. Tab and PALs should be deployed on the UTP. The integrity measurements of (attested) PALs and Tab should be provided to the client to enable verification.

Analysis. The table Tab fixes the set of identities of the PALs that are allowed to implement each part of the service functionality. When the client verifies the correctness of the last executed PAL, say $\mathtt{p_n}$, together with Tab, the client can trust that only valid PALs were used throughout the execution process. In fact, Tab ensures that only correct identities are used for secure storage operations, and only correct PALs have access to securely stored data that is critical for the execution.

D. Novel Secure Storage Solution

Secure channels to transfer intermediate results across trusted PAL executions should be available with low overhead. They should be (1) fast to set up, (2) require minimal TCC support, and (3) ensure mutual authentication of the end points.

The secure channel design described above can be built on current trusted components, but it is inefficient because its implementation provides more guarantees than desired. For example, on TPMs $v1.2$, sealed storage is based on asymmetric cryptography, which provides non-repudiation unnecessarily. As another example, while symmetric algorithms are available on TPMs $v2.0$, the trusted component still implements and enforces data access control (i.e., it checks whether the identified code is allowed to access the data), besides guaranteeing the confidentiality or integrity of sealed data. Intel SGX instead uses a different paradigm. The `ENCLU[EGETKEY]` instruction (for sealing) only provides a key to the Enclave based on its identity. The key is used by the Enclave to protect the data that can be then released outside its secure execution environment. Unfortunately, when two Enclaves need a shared secret key, they have to run an authentication protocol [7] to bootstrap trust in each other's attestations and validate public Diffie-Hellman keys. This involves at least two message exchanges, besides asymmetric cryptographic operations.

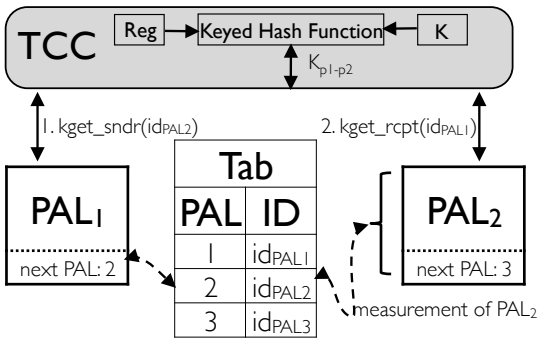

Fig. 6: Identity-based Secure Storage construction. It enables two PALs to share a mutually authenticated secret key in zero rounds, with no message exchange. Two PALs can use such key to transfer data with minimal overhead.

We propose a new construction that binds a secret shared key to the identities of two PALs efficiently. In particular, for any two PALs, the construction can build a secret key to create their secure channel. For any such key, only the PALs with the correct identity can access it. Our construction can be seen as a generalization of the Intel SGX approach, since a PAL is allowed to set up a secure channel not only with another code module but also with itself—e.g., to seal and save data in external untrusted storage. Last but not least, it is fast and requires minimal TCC support.

In contrast to TPM sealed storage, in our solution, the TCC does not make any access control decision on whether to accept or reject a PAL request, based on its current configuration (e.g., the value of the PCR registers) and the information included in the sealed data. The TCC always generates symmetric keys on-demand. For example, it is up to a PAL to decide to use the key to encrypt (or just authenticate) some result values and what code module can later retrieve this data. If an invalid module attempts to obtain encrypted data, it simply gets some random information because the wrong key is used for decryption. Similarly, if a valid module is run with incorrect data, it is simply unable to authenticate the initial input. Hence, it is essential that correct modules have access to correct identities.

Keys are derived from a master key `K`. This is a secret symmetric key that the TCC maintains internally for computing identity-dependent keys. Any PAL $\mathtt{p_i}$ can use an identity-dependent key to protect data. Any such key depends on: `K`, $\mathtt{p_i}$'s identity and another PAL $\mathtt{p_j}$'s identity. Also, the key can only be accessed by $\mathtt{p_i}$ and $\mathtt{p_j}$.

When module $\mathtt{p_i}$ wants to secure some information, it calls kget_sndr with the identity of the receiver $\mathtt{p_j}$. The TCC then performs the operations in Fig. 5 to derive a secret shared key $\mathtt{K}_{\mathtt{p_i}-\mathtt{p_j}}$ that is then returned to the PAL. To retrieve the same key at a later moment, $\mathtt{p_j}$ invokes kget_rcpt to perform an equivalent operation. Provided that the source and the recipient PALs respectively supply each other's identity (i.e., resp. $rcpt \equiv \mathtt{p_j}$ or $sndr \equiv \mathtt{p_i}$) to the TCC, the computed key is the same in the two cases.

The usage of the shared key for the new secure storage construction is shown in Fig. 6. It allows the protection of data to be transmitted between adjacent PALs. PALs can use the identity table Tab to look up the identity of the next executing PAL according to the control flow. Next, the sender PAL (resp. recipient PAL) calls kget_sndr (resp. kget_rcpt) to obtain the shared key. The key is then used by a function internal to

	Entity	fvTE Protocol
1	C → UTP	Request service execution with input **in** and nonce **N**
2	UTP	Prepare input: $\mathbf{in_1} \leftarrow \mathbf{in} \,\|\|\, \mathbf{N} \,\|\|\, \mathbf{Tab}$
3	UTP↔TCC	$\mathbf{\{\{out_1\}_{Kp1\text{-}p2}, Tab[1], Tab[2]\}} \leftarrow \mathbf{execute(p_1, in_1)}$
4		Repeat **for** $\mathbf{2 \le i \le n\text{-}1}$
5		$\mathbf{\{\{out_i\}_{Kpi\ -\ pi+1}, Tab[i], Tab[i+1]\}} \leftarrow$ $\mathbf{execute(p_i, \{out_{i-1}\}_{Kpi-1\ -\ pi} \,\|\| Tab[i\text{-}1])}$
6		$\mathbf{\{out_n, report\}} \leftarrow \mathbf{execute(p_n, \{out_{n-1}\}_{Kpn-1\ -\ pn} \,\|\| Tab[n\text{-}1])}$
7	UTP → C	Return to client: $\mathbf{\{out_n, report\}}$
8	C	Check execution: $\mathbf{verify(h(p_n), h(in) \,\|\| h(Tab) \,\|\| h(out_n), N, K^{+}_{TCC}, report)}$

	PAL	execute() step
9	p_1	Identify $\mathbf{p_1}$ in **REG**
10		Execute $\mathbf{p_1}$ with $\mathbf{in_1}$ and compute **out**
11		$\mathbf{out_1} \leftarrow \mathbf{out} \,\|\|\, \mathbf{h(in)} \,\|\|\, \mathbf{N} \,\|\|\, \mathbf{Tab}$
12		$\mathbf{\{out_1\}_{Kp1\text{-}p2}} \leftarrow \mathbf{auth\_put(Tab[2], out_1)}$
13		Return: $\mathbf{\{\{out_1\}_{Kp1\text{-}p2}, Tab[1], Tab[2]\}}$
14	p_i	Identify $\mathbf{p_i}$ in **REG**
15		$\mathbf{in_i} \leftarrow \mathbf{auth\_get(Tab[i\text{-}1], \{out_i\}_{Kpi\ -1\ -\ pi})}$
16		Execute $\mathbf{p_i}$ with $\mathbf{in_i}$ and compute **out**
17		$\mathbf{out_i} \leftarrow \mathbf{out} \,\|\|\, \mathbf{h(in)} \,\|\|\, \mathbf{N} \,\|\|\, \mathbf{Tab}$
18		$\mathbf{\{out_i\}_{Kpi\ -\ pi+1}} \leftarrow \mathbf{auth\_put(Tab[i+1], out_i)}$
19		Return: $\mathbf{\{\{out_i\}_{Kpi\ -\ pi+1}, Tab[i], Tab[i+1]\}}$
20	p_n	Identify $\mathbf{p_n}$ in **REG**
21		$\mathbf{in_n} \leftarrow \mathbf{auth\_get(Tab[n\text{-}1], \{out_{n-1}\}_{Kpn\ -1\ -\ pn})}$
22		Execute $\mathbf{p_n}$ with $\mathbf{in_n}$ and compute **out**
23		$\mathbf{out_n} \leftarrow \mathbf{out} \,\|\|\, \mathbf{h(in)} \,\|\|\, \mathbf{N} \,\|\|\, \mathbf{Tab}$
24		$\mathbf{report} \leftarrow \mathbf{attest(N, h(in) \,\|\| h(Tab) \,\|\| h(out_n))}$
25		Return: $\mathbf{\{out_n, report\}}$

Fig. 7: fvTE protocol run by client C, the trusted component TCC and the UTP (above, lines 1-8), and the execute step at the various PALs p_i (below, lines 9-25). A single attestation and verification allows the client to trust the service execution, despite the number of executed PALs. Also, only PALs that are necessary to serve a specific request are loaded and executed in the TCC.

the PAL to secure (resp. validate) the data to be released to (resp. supplied by) the UTP. In order to keep the terminology simple, we will henceforth reuse the names of the TCC secure storage primitives — auth_put and auth_get — to refer to these internal functions.

Analysis. In the execution of the key derivation function, the TCC uses the trusted identity of the currently executing PAL and a possibly untrusted identity provided by the PAL itself. These are positioned differently by the TCC in the f function, depending on whether the current PAL is saving or retrieving data (as in Fig. 5). The presence and the eventual verification of table Tab ensure that only correct identities (and thus PALs) are used to call the key derivation function. Furthermore, since a valid PAL forwards the data to the proper next PAL in accordance with the control flow, this guarantees that the right order of execution is followed and that only the correct next PAL can decrypt/validate the data.

E. A Flexible Trusted Execution Protocol

We now integrate the techniques discussed in the previous subsections into the *Flexible and Verifiable Trusted Execution (fvTE)* protocol detailed in Fig. 7. The protocol ensures all properties specified in Section II, namely it allows a client to securely and efficiently check the correctness of an arbitrary code execution. We now describe the main steps.

The client begins the protocol by submitting a service request to the UTP. It includes in the request a reference to the service input in plus a nonce N. The UTP then starts running the first PAL p_1 by providing the client's input, the nonce and the identity table, i.e., $< \mathtt{in}||\mathtt{N}||\mathsf{Tab} >$ (Lines 2-3). Notice that this is the only entry point of non-authenticated (and thus untrusted) data. However, the correctness of such data will be eventually verified by the client before accepting the reply.

The first PAL is run with the input and produces an intermediate state out (Lines 9-10). Before returning, it prepares the data to be forwarded to the next PAL: the output, a hash of the input, the nonce and the identity table (Line 11). The hash is used as an optimization to minimize the information to be transferred to subsequent PALs. This data is secured through auth_put, specifying the identity of the PAL that should follow in the execution flow ($p_2 = \mathsf{Tab}[2]$ [2]). The PAL terminates by providing to the UTP the secured intermediate state, and the identity of the current and the next PAL (Line 13).

The execution of the subsequent intermediate PALs proceeds similarly. They use auth_get to obtain the previous intermediate state (Line 15), whose validity derives from the properties of the secure channel. They execute their service code and propagate the result according to the expected control flow (Lines 17-19). Notice that values $< \mathtt{h(in)}||\mathtt{N}||\mathsf{Tab} >$ are simply left unchanged by each intermediate PAL as a way to propagate them to the final PAL ($p_n = \mathsf{Tab}[n]$).

p_n prepares the output for the client. After it retrieves the intermediate result from secure storage, it executes the code (Lines 21-22) and performs an attestation that binds together p_n's identity, the nonce, the client's request input, the identity table and the final output (Lines 23-24). When p_n terminates, it releases the final output and the report to the UTP (Line 25).

The UTP forwards the output to the client for verification (Line 7). At this point, the client has the following information: p_n's identity and $\mathtt{h}(\mathsf{Tab})$ that were outsourced by the authors of the code; the originally created request (input) and the fresh nonce; the final output and the attestation as issued by p_n; the trusted TCC's public key (see assumptions in Section III). The client can thus verify the attestation, and so the execution correctness, and trust the service output (Line 8).

Discussion. The protocol ensures that the properties in Section II-C are achieved as follows:

1) **Secure proof of execution**. The proof is unforgeable because it is conveyed by an attestation, i.e., a digital signature over the input, the output, the identity table (over secure hashes of these values). The signature is linked to the TCC hardware root of trust through a chain of digital certificates, whose ultimate root is a Certification Authority trusted by the client. The proof is unambiguous because of the attested identity of the last PAL p_n and Tab, and it is unique due to the inclusion of the nonce N. The execution flow cannot be tampered with, since only the correct PAL's can be run in the expected order. This last point is ensured through the novel storage primitive (and the identity table Tab) that prevents invalid PALs from accessing and tampering with the output of the intermediate states.
2) **Verification efficiency**. The client only has to perform

[2]Notice that "2" actually corresponds to the index of the next PAL in the execution flow that is hard-coded in p_1. The index is used for the lookup operation in Tab. We use this simplification in the description for brevity.

a constant number of hashes and check one digital signature to validate the result. Such verification effort is independent from the number of executed PALs.

3) **Communication efficiency**. The client interacts only once with the UTP to send the input `in` and receive the output `out` of the service. Also, the client provides and receives a constant additional amount of data, i.e., the nonce `N` and `report`.
4) **Low TCC resource usage**. Throughout the protocol, only the PALs that are required to serve the client request are loaded, identified and run. Furthermore, public key cryptography usage is limited to one attestation, while symmetric cryptography is used for fast key derivation on the TCC. Hence, the protocol consumes TCC resources efficiently and proportionally to what is actually executed.
5) **TCC-agnostic execution**. The execution protocol only uses the TCC abstraction. As the interface can be implemented on different trusted components, the protocol is not restricted to any specific architecture, so it is general. The next section explains one possible implementation.

Amortizing the attestation cost. Reducing the number of attestations provides benefits both to the UTP and to the client. However, as we show in our evaluation, a single attestation can still be computationally expensive when the client has to verify multiple requests. It is common practice to avoid this overhead by setting up a secure session, between the trusted environment and the client, based on a symmetric secret key. We sketch a possible solution using our protocol.

The code base can be enriched with another PAL, $\mathtt{p_C}$, that establishes the secure channel. $\mathtt{p_C}$ receives the client's fresh public key $\mathtt{pk_C}$ as input at the beginning of the computation. It assigns the identity $\mathtt{id_C} = \mathtt{h}(\mathtt{pk_C})$ to the client; it uses kget_sndr (§IV-D) to retrieve the identity-dependent key $\mathtt{K_{p_C-C}}$ (to be) shared with the client; it encrypts $\mathtt{K_{p_C-C}}$ with $\mathtt{pk_C}$. The attestation of the result and the encrypted data are sent back to the client. The client verifies the attestation and retrieves $\mathtt{K_{p_C-C}}$. In subsequent requests, the client authenticates (or encrypts) messages with $\mathtt{K_{p_C-C}}$ and attaches $\mathtt{id_C}$. The client's identity allows $\mathtt{p_C}$ to recompute $\mathtt{K_{p_C-C}}$ without maintaining any session state. $\mathtt{p_C}$ can thus authenticate the message and forward it to the first PAL in the original execution flow. Similarly, $\mathtt{p_C}$ should receive the computed reply from the last PAL so to build an authenticated message for the client.

V. Experimental Analysis

This section focuses on the implementation and evaluation of our protocol when applied to a real-world service. The protocol is used to securely link together code modules of the widely-deployed SQLite database engine. A formal verification of the correctness is carried out with Scyther. Our results show that the code identification overhead can be significantly reduced without trading off security and functionality.

A. Implementation

Trusted component. We implemented the TCC using XMHF/TrustVisor [32, 46], which is based on a hardware TPM, and whose code is open-source and easy to customize.

XMHF/TrustVisor is a security hypervisor that can perform trusted executions on-demand. Briefly, a trusted execution involves three steps, all initiated from the untrusted environment, where the OS and other services run. The hypervisor performs the following operations: it protects the memory regions of a PAL from external access and measures its code (PAL *registration* step); it executes the PAL and handles the marshaling of I/O parameters between the trusted and the untrusted environment (PAL *execution* and *termination* step); it clears the PAL's state and makes it accessible in the untrusted environment (PAL *unregistration* step).

In order to implement our protocol, we modified XMHF/TrustVisor by adding three hypercalls. The first makes memory available to a PAL in its address space. This avoids allocating memory in the untrusted environment, then transferring it to the trusted environment and making it accessible to a PAL as dynamic memory. Consequently, such memory space is neither part of a PAL's identity, nor of a PAL's input data, and it can be provided more efficiently. The second hypercall is kget_sndr, which is used in the auth_put primitive to retrieve a shared key to secure some data for a known receiver PAL. As the TCC only computes a secret key, this allows a developer to choose and implement the security technique that is most suitable for the application (e.g., message authentication codes or authenticated encryption). The third hypercall is kget_rcpt, which is used in the auth_get primitive to retrieve a shared key to validate some data that was previously secured by a known sender PAL. The TCC-specific key used for identity-dependent key derivation is initialized inside XMHF/TrustVisor when the platform boots.

Platform. We used a Dell PowerEdge R420 Server, with a 2.2GHz Intel Xeon E5-2407 CPU, 3GB of memory, a TPM v1.2, and running Ubuntu 12.04 with a Linux kernel 3.2.0-27. The resources were fully dedicated to our experiments.

Application. Our protocol was applied to the SQLite database engine [40], which has a code base of about $88K$ lines of source code. SQLite is open-source and widely deployed, e.g., on Android [4], iCloud [5] and other operating systems [3].

A multi-PAL SQLite engine was created with a small per-operation code footprint. Different PALs were built to handle specific queries. Each one was handcrafted by trimming the unused code off the original code base. Then, our protocol is used to securely link these PALs together.

Our current multi-PAL SQLite engine consists of 4 PALs that implement some of the most representative SQL operations. We emphasize that additional operations can be included by following the same approach—see Section VII—so to match the functionality of the original database engine. PAL_0 is the first one called from the untrusted environment on the UTP and it receives the input data from the client. PAL_0 parses the client's request to recognize the type of query, and then forwards it to a specialized PAL for the execution by means of our secure channels. $Select$ queries are passed to PAL_{SEL}. $Insert$ queries are sent to PAL_{INS}. $Delete$ queries are passed to PAL_{DEL}. The last executed PAL builds the reply that is released to the UTP's untrusted environment, from which it is then forwarded to the client.

We compare multi-PAL SQLite against a baseline implementation of the full SQLite database engine. We implemented it as a monolithic PAL_{SQLITE} that can execute any query.

We perform end-to-end experiments, where a client performs $select$, $insert$ and $delete$ queries on the server that maintains a database. Queries are received through a ZeroMQ [50] socket at the UTP, and delivered to PAL_0 for

initial processing. The experiments were based on a small size database because it highlights the overhead due to code identification, which is the focus of this paper.

Execution Flows. In multi-PAL SQLite, requests from the client follow the following execution flows: $PAL_0 \rightarrow PAL_{SEL}$, $PAL_0 \rightarrow PAL_{INS}$, or $PAL_0 \rightarrow PAL_{DEL}$. Any other query is currently discarded by PAL_0 and the trusted execution terminates. However, additional operations could be easily using the same procedure.

B. Correctness

We verified the correctness of the $fvTE$ protocol applied to SQLite using Scyther [14, 15], a public tool for the formal verification of security protocols. We chose Scyther as it supports unbounded verification of security properties or their violation by providing feasible attacks.

Protocol Modeling. The security protocol is performed among the following entities: the client, the 4 PALs and the TCC. The UTP is untrusted and it is modeled by Scyther as an adversary that is able to forge and replay messages. We describe the execution verification of a *select* query (but it will be evident that it can be adapted to other executions in a straightforward manner).

Messages are exchanged on two channels: one between the client and the TCC; and another between the TCC and a PAL that is executing. The first one is modeled as an insecure channel because the client and the TCC do not share any secret. The first message is therefore not secured and the last message is signed by the TCC (i.e., attested through $\mathtt{K}_{\mathtt{TCC}}^{-}$). The second channel is instead modeled as a secure channel. We let the TCC and each PAL i share a fresh secret key (e.g., $K_{TCC \leftrightarrow PAL_i} \equiv K_{PAL_i \leftrightarrow TCC}$) to secure their communication. The reason is that each PAL runs (and terminates) above the TCC when the execution environment is already isolated. This implies a secure data/control transfer between the TCC and each PAL.

A logical secure channel is available between pairs of PALs. The channel is protected with the key (for instance, $K_{PAL_0 \leftrightarrow PAL_{SEL}}$) shared between the indicated PALs. The TCC essentially forwards messages between the direct channels that it establishes with each PAL. This is modeled through message encapsulation: a PAL first secures the message using the key that it shares with another PAL, and then it secures the message again using the key shared with the TCC. The security of the channel derives from our construction in Section IV-D.

Protocol Verification. The execution chain is verified in three steps. First, the TCC validates that PAL_0 successfully completes an execution on inputs Req, N, Tab and delivers a response Res_{PAL_0} securely linked to the inputs. This allows the TCC to trust that Res_{PAL_0} is the correct output of PAL_0. Second, the TCC validates that PAL_{SEL} successfully completes an execution on inputs $Res_{PAL_0}, h(Req), N, Tab$ and delivers a response $Res_{PAL_{SEL}}$ securely linked to the inputs. This lets the TCC trust that $Res_{PAL_{SEL}}$ is the correct output of PAL_{SEL}. Third, the client validates that the TCC successfully completes an execution on inputs Req, N, Tab and delivers a response $Res_{PAL_{SEL}}$ securely linked to the inputs. Finally, this allows the client to trust that $Res_{PAL_{SEL}}$ is the valid output.

Scyther verified the protocol execution in about 35 minutes, on a MacBook Pro with a 2.3GHz Intel i7 CPU.

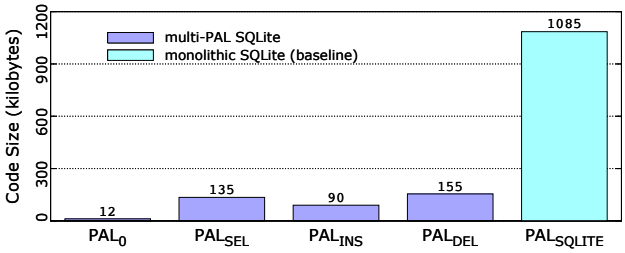

Fig. 8: Size of each PAL's code in our SQLite code base.

Discussion. The reader should note that the successful verification refers to the $fvTE$ protocol as applied to the multi-PAL SQLite design and not to the general protocol (in Fig. 7). However, this verification together with the analysis performed during the protocol description (Section IV-E) gives us confidence that our approach is correct. Verifying an actual implementation is an orthogonal problem that could be addressed with Ironclad Apps [22].

C. Evaluation

We evaluate the multi-PAL SQLite and compare it against the full monolithic SQLite. An always-positive speed-up was observed with our design, which shows that for this setting it is convenient to load and integrity-measure only the modules that are executed out of a large code base.

Code Size. The size of the code for each PAL protected by XMHF/TrustVisor at registration time is shown in Fig. 8. The size of the full SQLite implementation is about $1MB$, while common operations such as $select$, $insert$, $delete$ can be implemented in as little as 9-15% of the code base.

speed-up	W/ ATTESTATION	W/O ATTESTATION
INSERT	1.46×	2.14×
DELETE	1.26×	1.63×
SELECT	1.32×	1.73×

TABLE I: Summary of the achieved per-operation speed-up.

End-to-end performance. The performance results for each execution flow are displayed in Fig. 9, and summarized in Table I. Each run is one end-to-end query execution, i.e., the client sends one request and receives the corresponding reply. We have included the execution times with and without attestation. The average of at least 10 runs is displayed with the 95% confidence interval. XMHF/TrustVisor computes an attestation using a $2048bit$ RSA key and, in our testbed, it takes around $56ms$. Such overhead could be reduced by establishing a secure session with the client (see §IV-E).

Overall, our protocol improves substantially on the previous approach. For example, $insert$ is about 1.46× faster than the traditional approach using the monolithic SQLite; the result could be improved to become up to 2× faster by considering more efficient attestation mechanisms. Notice that if the original code base gets larger, then the benefit increases.

At the application level (i.e., without considering the underlying TCC overhead), the execution time of SQLite is similar for queries that are executed in the monolithic PAL_{SQLITE} or in the small PALs. This is expected since they execute essentially the same code on the same state. Consequently, the performance differences are mainly the result of the different size of the code that is loaded in the trusted environment.

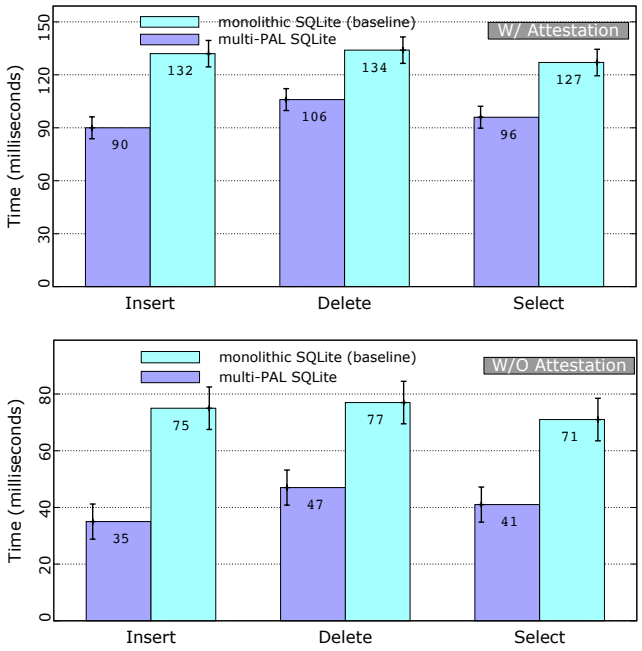

Fig. 9: Performance comparison between the multi-PAL and the monolithic SQLite databases.

Finally, we measured the overhead of PAL_0 in our end-to-end experiments. PAL_0 terminates its execution in about $6ms$. Considering attestation, this corresponds to an overhead of 6.6% for $insert$, 5.6% for $delete$, 6.2% for $select$. Without attestation, the overhead is 17.1%, 12.7%, 14.6% respectively.

Optimized vs. non-optimized secure channels. We compare our secure storage construction (Section IV-D) with the original one of XMHF/TrustVisor (i.e., *seal* and *unseal*). Both use symmetric cryptography, but XMHF/TrustVisor's secure storage requires more operations for: (i) managing TPM-like data structures because it implements a software micro-TPM; (ii) using AES for encryption, in order to guarantee secrecy of sealed data; (iii) retrieving random numbers for the initialization vector to guarantee semantic security; (iv) using SHA1-HMAC for integrity protection. Instead, our construction only uses SHA1-HMAC, keyed with the TCC secret created at boot time, to derive identity-dependent keys.

The results of the performance measured inside the hypervisor are: $15\mu s$ and $16\mu s$ for kget_rcpt and kget_sndr; and $122\mu s$ and $105\mu s$ for *seal* and *unseal* respectively. The operations in our construction are respectively $8.13\times$ and $6.56\times$ faster. In our experiments, using XMHF/TrustVisor's native secure storage (recall from Section IV-D that both can be used to implement secure channels) does not change the results in Fig. 9 noticeably. The difference in overhead is at least two orders of magnitude smaller than the end-to-end execution time. Notice however that in large-scale services of several interconnected PALs and long execution flows, such overhead could become non-negligible.

VI. Performance Model for Code Identification

In this section we devise a performance model for code identification to study under what circumstances using the *fvTE* protocol outperforms the traditional approach of monolithic trusted executions. For the traditional approach, we can model

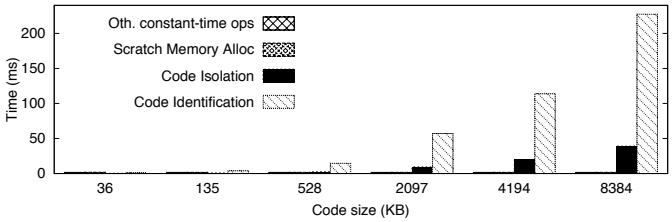

Fig. 10: Breakdown of the code registration costs inside XMHF/TrustVisor.

the costs for code execution as follows:

$$T = \underbrace{(t_{is}(\mathcal{C}) + t_{id}(\mathcal{C}) + t_1)}_{\text{code protection cost}} + \underbrace{(t_{is}(in) + t_{id}(in) + t_2)}_{\text{input protection cost}} + \underbrace{(t_{is}(out) + t_{id}(out) + t_3)}_{\text{output protection cost}} + \underbrace{t_{att}}_{\text{attestation cost}} + \underbrace{t_X}_{\text{execution cost}}$$

(TCC-dependent costs)

We distinguish between TCC-related costs and application-level costs. The latter (t_X) is invariant with respect to the trusted execution protocol actually used, and only depends on the platform where the application runs. The former instead depends on the TCC and on the implemented protocols for isolation (is), identification (id) and attestation (att) of a code base ($\mathcal{C}$) and input/output (in/out) data. As shown later, identification and isolation costs are linear in the size of the argument ($\mathcal{C}$, in, or out), while t_1, t_2, t_3 are constant additional costs—so linear costs are modeled as $y = ax + bx + c$.

The code protection cost thus impacts part of the overall cost for a trusted execution. Such an impact is less noticeable when the input/output data protection costs or the execution cost outweigh the code protection cost. However, the focus of this paper is on code identification. Therefore, for the sake of performance modeling, we put emphasis on trusted executions where the code protection cost outweighs the other terms with the following approximation

$$T \approx t_{is}(\mathcal{C}) + t_{id}(\mathcal{C}) + t_1$$

The experimental quantification of these costs in XMHF/TrustVisor is shown in Fig. 10. We built a set of PALs each having an increasing number of `NOP` operations. The times for code isolation and identification grow with code size. Other operations, including scratch memory allocation, are code-independent and have constant cost (i.e., t_1 overall).

We model the costs of the *fvTE* protocol in a similar way:

$$T_{fvTE} = (t_{is}(\mathcal{E}) + t_{id}(\mathcal{E}) + nt_1) + n\,(t_{is}(in) + t_{id}(in) + t_2) + n\,(t_{is}(out) + t_{id}(out) + t_3) + t_{att} + t_X$$

Here $\mathcal{E}$ is the set of n PALs in an execution flow, and we define $|\mathcal{E}|$ as their aggregated size. Code protection costs are approximated as—notice the per-PAL constant costs:

$$T_{fvTE} \approx t_{is}(\mathcal{E}) + t_{id}(\mathcal{E}) + nt_1$$

Our protocol is more efficient than the previous approach when protecting the execution flow is less expensive than protecting the whole code base. This can be defined as:

$$\underset{ratio}{efficiency} \quad \frac{T}{T_{fvTE}} \begin{cases} \text{positive, if } > 1 \\ \text{negative, if } \leq 1 \end{cases}$$

A positive efficiency ratio indicates that it is worth having

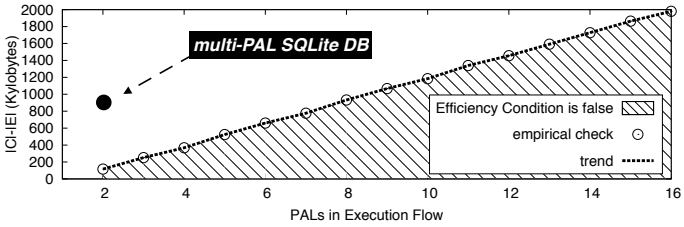

Fig. 11: Validation of the performance model. The slope of trend line represents the t_1/k constant in the efficiency condition. $|\mathcal{C}|$ is the size of the code base. $|\mathcal{E}|$ is the size of the code in an execution flow.

multiple PALs. Instead, a negative efficiency ratio indicates that it is better to protect the whole code base. The (positive) efficiency condition can be defined as follows. First, given the linearity of the code isolation and identification costs, we group them as $t_{id}(\mathcal{C}) + t_{is}(\mathcal{C}) = k|C|$, for some constant k. Then:

$$\frac{k|\mathcal{C}| + t_1}{k|\mathcal{E}| + nt_1} > 1 \quad \rightarrow \quad \frac{|\mathcal{C}| - |\mathcal{E}|}{n-1} > \frac{t_1}{k} \quad \begin{matrix} efficiency \\ condition \end{matrix}$$

The efficiency ratio depends on both the size of the code base and the size of the execution flow. However, the efficiency condition depends only on their difference (in addition to n and t_1/k, the architecture-specific constant discussed later).

We validate the model through an experiment that uses different sets of PALs with cardinality from 2 to 16. For each set we varied the aggregated size $|\mathcal{E}|$ (i.e., the size of executed code). We empirically measured the maximum aggregated size for each set for which the *fvTE* protocol is faster than the traditional monolithic approach. This corresponds to the *empirical check* in Fig. 11. Notice that the trend of these empirical measurements is well approximated by a straight line which divides the plane in two areas: the shaded one where the efficiency condition is false, and the other area where it is true. The slope of the line represents the constant t_1/k.

Discussion. The constant t_1/k depends strongly on the TCC. In our experiments, it depends on our testbed hardware platform and the software (XMHF/TrustVisor, see Section V-A) that provides trusted computing services. In Flicker [33] both terms are larger due to the interaction with the slow TPM, particularly k for the identification. Instead, future trusted computing technologies such as Intel SGX [25] are expected to reduce significantly both t_1 and k. However, since the constant also depends on the software that supports trusted executions, it is difficult to predict its trend without running experiments on a real platform.

VII. Related Work

Code identity and trusted executions. Code identity has been originally defined as the digest of a program's code in [30]. The same definition was later borrowed for trusted hardware-based code executions [17, 18] as a useful mechanism for sealed storage and attestation purposes. Current platforms and CPU extensions such as AEGIS[41], Intel TXT [24], Intel SGX [25], OASIS [36], TrustLite [28] allow to identify some code before the execution by hashing its content. Tools that leverage some of these architectures, such as Flicker [33], TrustVisor [32], Haven [10] do not address the problem of code size inside the trusted environment and execute monolithic applications, whose identity can be verified remotely. In this paper, we do not change the definition of code identity (i.e., the hash of the binary), and we observe that another way for the client to verify a remote execution is to (be able to) make trust inferences. Therefore, by building a robust chain of trust throughout the modules of a large code base, it is sufficient for the client to verify only part of the chain to infer that the execution of the whole code base was performed correctly.

OASIS [36] proposes to deal with an application whose size is greater than the cache by building a Merkle tree over its code blocks. However, it requires new hardware support, so it does not provide general solution that retrofits existing trusted computing components. Our protocol instead could leverage OASIS by implementing our TCC abstraction (Section III) and, with minimal modifications, it could also include our novel secure storage construction (Section IV-D).

The BIND service [38] leverages fine-grained code attestation to secure a distributed system. BIND targets small pieces of code, while our protocol is able to provide execution integrity guarantees of large code bases. Additionally, although small modules could use BIND to build a chain by verifying each other, the resulting construction (similar to that in Section IV-A) would not be verification efficient and could incur verification loop issues (Section IV-C). Our protocol addresses these drawbacks and guarantees integrity when the client eventually verifies the execution. Finally, BIND's security kernel was not implemented [33].

A research work related to ours is the On-board Credential (ObC) Project [16, 29, 12]. The ObC Project defines an open architecture based on secure hardware [43, 44] for the installation and execution of credential mechanisms on constrained ObC-ready (typically mobile) devices. It enables a service provider to provision secrets to a family of (installed) credential programs [29], which are executed slice-wise in a secure environment [16], possibly using the TPM's late launch mechanism [12]. Such credential programs are application or platform-specific, while our work is concerned about the efficient verification of executions that are performed on a generic trusted component. The chain of trust among the slices is based on the *slice endorsement token*, containing the family and program-specific secrets, which is created *online* on a per-slice basis. In our case, the chain is explicit in each PAL through a reference to the previous/next PAL's identity, and only needs an *offline* setup (i.e., the process of making the code base available on the UTP) performed by the service authors. Also, access to secured data is controlled by (our) construction through the trusted component, allowing secure data exchange among PALs pairwise.

Defining code modules. Making modules/partitions out of programs is a programming-language problem that has been widely studied, e.g., in the context of privilege separation [27], parallel program execution [19, 48] and secure distributed computation [49]. Defining such a method for PALs is orthogonal to and out of the scope of this paper. We mention however that we built our SQLite-based prototype (Section V-C) by using both static and dynamic program analysis to distinguish the non-active code and remove it, and performing extensive testing to check the correctness of the resulting active code. As an additional example, in another application for secure image filtering, we implemented and protected each filter as a separate task, and then created a secure and efficiently verifiable chain using our protocol, though a different TCC.

VIII. Conclusions

In this paper we have shown that current trends in Trusted Computing create a trade-off between security and efficiency due to code identity assignment. We presented a general protocol that enables efficiently verifiable (at the client) and flexible (at the UTP) trusted executions of arbitrarily sized code bases by identifying only the actively executed code. We successfully applied our protocol to a real-world database engine, showing positive results already with a 1MB code base.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was partially supported by the EC through project H2020-643964 (SUPERCLOUD), by national funds of Fundação para a Ciência e a Tecnologia (FCT) through project UID/CEC/00408/2013 (LaSIGE), and by the research grant SFRH/BD/51562/2011.

References

[1] "Exploit-DB," *http://www.exploit-db.com/*.

[2] "Open Sourced Vulnerability Database," *http://www.osvdb.org/*.

[3] "SQLite Deployments," *http://sqlite.org/mostdeployed.html*.

[4] "SQLite in Android," *http://developer.android.com/reference/ android/ database/ sqlite/ SQLiteDatabase.html*.

[5] "SQLite in iCloud," *https://developer.apple.com/library/ios/ documentation/ DataManagement/ Conceptual/ UsingCoreData WithiCloudPG/ UsingSQLiteStoragewithi Cloud/UsingSQLiteStoragewithiCloud.html*.

[6] Amazon, "AWS CloudHSM, Secure Key Storage and Cryptographic Operations," *http://aws.amazon.com/cloudhsm*.

[7] I. Anati and S. Gueron, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[8] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th Conference on Computer and Communications Security (CCS)*, 2011, pp. 375–388.

[9] S. Bajaj and R. Sion, "TrustedDB," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2011, pp. 205–216.

[10] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, oct 2014, pp. 267–283.

[11] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "Trusted Computing - Challenges and Applications," in *Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies (TRUST)*, vol. 4968, 2008, pp. 14–32.

[12] S. Bugiel and J.-E. Ekberg, "Implementing an application-specific credential platform using late-launched mobile trusted module," in *Proceedings of the 5th ACM workshop on Scalable trusted computing (STC)*, 2010, p. 21.

[13] B. Chen and R. Morris, "Certifying program execution with secure processors," in *Proceedings of the 9th conference on Hot Topics in Operating Systems (HOTOS)*, may 2003, p. 23.

[14] C. J. Cremers, "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols," in *Proceedings of the 20th international conference on Computer Aided Verification (CAV)*, vol. 5123, Berlin, Heidelberg, jul 2008, pp. 414–418.

[15] ——, "Unbounded verification, falsification, and characterization of security protocols by pattern refinement," in *Proceedings of the 15th conference on Computer and Communications Security (CCS)*, oct 2008, p. 119.

[16] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala, "Scheduling execution of credentials in constrained secure environments," in *Proceedings of the Workshop on Scalable Trusted Computing (STC)*, 2008.

[17] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[18] P. England and M. Peinado, "Authenticated Operation of Open Computing Devices," in *Proceedings of the 7th Australian Conference on Information Security and Privacy (ACISP)*, jul 2002, pp. 346–361.

[19] M. Girkar and C. Polychronopoulos, "Partitioning programs for parallel execution," in *Proceedings of the 2nd Int. Conference on Supercomputing (ICS)*, 1988, pp. 216–229.

[20] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, apr 2009.

[21] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the 3rd ACM workshop on Scalable Trusted Computing (STC)*, 2008, pp. 11–20.

[22] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: end-to-end security via automated full-system verification," in *Proc. of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, oct 2014, pp. 165–181.

[23] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag," *ACM SIGPLAN Notices*, vol. 48, no. 4, p. 265, apr 2013.

[24] Intel, "Intel Trusted Execution Technology," *http://www.intel.com/ content/dam/www/ public/us/en/ documents/ guides/ intel-txt-software-development-guide.pdf*.

[25] ——, "Software Guard Extensions," *https://software.intel.com/sites/ default/files/managed/48/88/329298-002.pdf*.

[26] B. Kauer, "OSLO: improving the security of trusted computing," in *Proceedings of 16th USENIX Security Symposium*, aug 2007, p. 16.

[27] D. Kilpatrick, "Privman: A library for partitioning applications," in *Proceedings of the USENIX Annual Technical Conference (Freenix track)*, 2003.

[28] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014, pp. 1–14.

[29] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009, p. 104.

[30] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: theory and practice," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 265–310, nov 1992.

[31] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: a two-way sandbox for x86 native code," in *Proc. of the USENIX Annual Technical Conference (ATC)*, jun 2014, pp. 409–420.

[32] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation." in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 143–158.

[33] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proceedings of the European Conference in Computer Systems (EuroSys)*, vol. 42, no. 4, 2008, pp. 315–328.

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013, pp. 1–1.

[35] Microsoft, "BitLocker," *http://windows.microsoft.com/ en-us/ windows7/ products/ features/bitlocker*.

[36] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan, "OASIS," in *Proceedings of the 2013 Conference on Computer & Communications Security (CCS)*, nov 2013, pp. 13–24.

[37] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th USENIX Security Symposium*, 2004, p. 16.

[38] E. Shi, A. Perrig, and L. van Doorn, "BIND: A Fine-Grained Attestation Service for Secure Distributed Systems," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, may 2005, pp. 154–168.

[39] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks*, vol. 31, no. 9, pp. 831–860, 1999.

[40] SQLite, "http://www.sqlite.org/."

[41] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS," in *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, 2003, pp. 160–171.

[42] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 25–36.

[43] Trusted Computing Group, "MTM Specification v1.0 rev. 7.02," 2010.

[44] ——, "Mobile Trusted Module 2.0 Use Cases," 2011.

[45] ——, "TPM Main Specification Version 1.2, Rev. 116," 2011.

[46] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, may 2013, pp. 430–444.

[47] B. Vavala, N. Neves, and P. Steenkiste, "Securing Passive Replication Through Verification," in *Proceedings of the 34st IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2015, pp. 176–181.

[48] E. Yardimci and M. Franz, "Mostly static program partitioning of binary executables," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 5, pp. 1–46, jun 2009.

[49] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, pp. 283–328, aug 2002.

[50] ZeroMQ, "http://zeromq.org/."