Secure Multiparty Computation from SGX*

Raad Bahmani[1], Manuel Barbosa[2], Ferdinand Brasser[1], Bernardo Portela[2], Ahmad-Reza Sadeghi[1], Guillaume Scerri[3], and Bogdan Warinschi[4]

[1] Technische Universität Darmstadt
[2] HASLab – INESC TEC & DCC-FCUP
[3] Laboratoire DAVID – Université de Versailles St-Quentin & INRIA
[4] University of Bristol

Abstract. In this paper we show how Isolated Execution Environments (IEE) offered by novel commodity hardware such as Intel's SGX provide a new path to constructing general secure multiparty computation (MPC) protocols. Our protocol is intuitive and elegant: it uses code within an IEE to play the role of a trusted third party (TTP), and the attestation guarantees of SGX to bootstrap secure communications between participants and the TTP. The load of communications and computations on participants only depends on the size of each party's inputs and outputs and is thus small and independent from the intricacies of the functionality to be computed. The remaining computational load– essentially that of computing the functionality – is moved to an untrusted party running an IEE-enabled machine, an attractive feature for Cloud-based scenarios.

Our rigorous modular security analysis relies on the novel notion of labeled attested computation which we put forth in this paper. This notion is a convenient abstraction of the kind of attestation guarantees one can obtain from trusted hardware in multi-user scenarios.

Finally, we present an extensive experimental evaluation of our solution on SGX-enabled hardware. Our implementation is open-source and it is functionality agnostic: it can be used to securely outsource to the Cloud arbitrary off-the-shelf collaborative software, such as the one employed on financial data applications, enabling secure collaborative execution over private inputs provided by multiple parties.

1 Introduction

Secure multiparty computation (MPC) allows a set of mutually distrusting parties to collaboratively carry out a computation that involves their private inputs. The security guarantee that parties get are essentially those provided by carrying out the same computation using a Trusted Third Party (TTP). The computations to be carried out range from simple functionalities, for example where a party commits to a secret value and later on

* This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE). Manuel Barbosa and Bernardo Portela were funded by project "NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016", which is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

reveals it; or they can be highly complex, for example running sealed bid auctions [9] or bank customer benchmarking [17]. Most of the existent approaches are software only. The trust barrier between parties is overcome using cryptographic techniques that permit computing over encrypted and/or secret-shared data [31,25,18]. Another approach first studied by Katz [28] formalizes a trusted hardware assumption— where users have access to tamper-proof tokens on which they can load arbitrary code—that is sufficient to bootstrap universally composable MPC.

Broadly speaking, this work fits within the same category as that by Katz [13]. However, our starting point is a novel real-world form of trusted hardware that is currently shipped on commodity PCs: Intel's Software Guard Extensions [27]. Our goal is to leverage this hardware to significantly reduce the computational costs of practical secure computation protocols. The main security capability that such hardware offers are Isolated Execution Environments (IEE) – a powerful tool for boosting trust in remote systems under the total or partial control of malicious parties (hijacked boot, corrupt OS, running malicious software, or simply a dishonest service provider). Specifically, code loaded in an IEE is executed in isolation from other software present in the system, and built-in cryptographic attestation mechanisms guarantee the integrity of the code and its I/O behaviour to a remote user.

PROTOCOL OUTLINE. The functionality outlined above suggests a simple and natural design for *general multiparty computation*: load the functionality to be computed into an IEE (which plays the role of a TTP) and have users provide inputs and receive outputs via secure channels to the IEE. Attestation ensures the authenticity of the computed function, inputs and outputs. The resulting protocol is extremely efficient when compared to existing solutions that do not rely on hardware assumptions. Indeed, the load of communications and computations on protocol participants is small and independent of the intricacies of the functionality that is being computed; it depends only on the size of each party's inputs and outputs. The remaining computational load — essentially that of computing the functionality expressed as a transition function in a standard programming language — is moved to an untrusted party running an IEE-enabled machine. This makes the protocol attractive for Cloud scenarios. Furthermore, the protocol is non-interactive in the sense that each user can perform an initial set-up, and then provide its inputs and receive outputs independently of other protocol participants, which means that it provides a solution for "secure computation on the web" [24] with standard MPC security.

Due to its obvious simplicity, variations of the overall idea have been proposed in several practice-oriented works [36,23]. However, currently there is no thorough and rigorous analysis of the security guarantees provided by this solution in the sense of a general approach to MPC. The intuitive appeal of the protocol obscures multiple obstacles in obtaining a formal security proof, including: i. the lack of private channels between the users and the remote machine; ii. the need to authenticate/agree on a computation in a setting where communication between parties is inherently asynchronous and only mediated by the IEE; iii. the need to ensure that the "right" parties are engaged in the computation; iv. dealing with the interaction between different parts of the code that coexist within the same IEE, sharing the same memory space, each potentially corresponding to different users; and v. ensuring that the code running inside an IEE does not leak sensitive information to untrusted code running outside.

In this paper we fill this gap through the following contributions: i. a rigorous specification of the protocol for general MPC computation outlined above; ii. formal security definitions for the security of the overall protocol and that of its components;[5] iii. a modular security analysis of our protocol that relies on a novel notion of labelled attested computation; and iv. an open-source implementation of our protocol and a detailed experimental analysis in SGX-enabled hardware. We give an overview of our results next.

LABELED ATTESTED COMPUTATION. Our protocol relies on ideal functionalities viewed as programs written as transition functions in a programming language compatible with the IEE-enabled machine. We instrument these programs to run inside an IEE and add bootstrapping code that permits protocol participants to establish independent secure channels with the functionality, so that they can provide inputs and receive outputs. The crux of the protocol is a means to provide attestation guarantees which ensures that parties are involved in the "right" run of the protocol (i.e. with the right parties all interacting with the same IEE). We take inspiration from the recent work of Barbosa et al.[4] who provide a formalization for the notion of *attested computation* that can convince a party that its local view of the interaction with a remote IEE matches what actually occurred remotely. This guarantee is close to the one that we need, but it is unfortunately insufficient. The problem is that attested computation a la [4] is concerned with the interaction between a single party and an IEE, and it is non-trivial to extend these guarantees to the interaction of multiple parties with the same IEE when the goal is to reason about *concurrent asynchronous interactions*.

To overcome these problem, we introduce the notion of *labelled attested computation* (LAC), a powerful and clean generalization of the attested computation notion in [4]. In a nutshell, this notion assumes that (parts of) the code loaded in an IEE is marked with labels pertaining to users, and that individual users can get attestation guarantees for those parts of the code that corresponds to specific labels. The gain is that users can now be oblivious of other user's interactions with the IEE, which leads to significantly more simple and efficient protocols. Nonetheless, the user can still derive attestation guarantees about the overall execution of the system, since LAC binds each users' local view to the *same code* running within the IEE, and one can use standard cryptographic techniques to leverage this binding in order to obtain *indirect* attestation guarantees as to the honest executions of the interactions with other users.

We provide syntax and a formal security model for LAC and show how this primitive can be used to deploy arbitrary (labelled) programs to remote IEEs with flexible attestation guarantees. Our provably secure LAC protocol relies on hardware equipped with SGX-like IEEs. Our construction of the MPC protocol then builds on LACs in a modular way. First, we show how to use labelled attested computation schemes[6] to bootstrap an arbitrary number of independent secure channels between local users and

[5] Since our emphasis is on efficiency and analysing SGX-based protocols used in practice, we do not consider Universal Composability, but rather a simulation-based security model akin to those used for other practical secure computation protocols, e.g. [6].

[6] We use schemes which satisfy the additional notion of *minimal leakage* which ensures that the outsourced instrumented program P^* reveals no information about its internal state beyond what the normal input/output behavior of the original program P would reveal.

an IEE with joint attestation guarantees. We formalize this result as an *utility theorem*. The security of the overall MPC protocol which uses these channels for communication with functionality code inside an IEE is then built on this utility theorem.

IMPLEMENTATION AND EXPERIMENTAL VALIDATION. We conclude the paper with an experimental evaluation of our protocol via a detailed comparison of our solution to state-of-the-art multiparty computation. The experimental results confirm the theoretical performance advantages that we have highlighted above in comparison to non hardware-based solutions. Our implementation of a generic MPC protocol —sgx-mpc— relies on the NaCl[7] cryptographic library [8] and inherits its careful approach to dealing with timing side-channels. We discuss side-channels in SGX-like systems and explain how our *constant-time* code thwarts *all* leaks based on control-flow or memory access patterns that depend on secret data.

Our implementation is functionality agnostic and can be used to outsource to the Cloud arbitrary off-the-shelf collaborative software, enabling multiple parties to jointly execute complex interactive computations without revealing their own inputs. Taking the financial sector as an example, our implementation permits carrying out financial benchmarking [17] using off-the-shelf software, rather than requiring the conversion of the underlying computation into circuit form, as is the case in state-of-the-art secure multiparty computation protocols. One should of course mention that, in order to meet the level of side-channel attack resilience of sgx-mpc, the code that is outsourced to the Cloud should itself be implemented according to the constant-time coding policy. This, however, is a software engineering issue that is outside of the scope of this paper.

RELATED WORK. A relevant line of research leverages trusted hardware to bootstrap entire platforms for secure software execution (e.g. Flicker [32], Trusted Virtual Domains [14], Haven [5]). These are large systems that are currently outside the scope of provable-security techniques. Smaller protocols which solve specific problems are more susceptible to rigorous analysis. Examples of these are secure disk encryption [33], one-time password authentication [26] outsourced Map-Reduce computations [36], Secure Virtual Disk Images [22], two-party computation [23], secure embedded devices [34,29]. Although some of these protocols (e.g., those of Hoekstra et al. [26] and Gupta et al [23]) come only with intuition regarding their security, others—most notably those by Schuster et. al [36]—come with a proof of security. The use of attestation in those protocols is akin to our use of attestation in our general MPC protocol. Provable security of realistic protocols that use trusted hardware-based protocols based on the Trusted Platform Module (TPM) have been considered in [11,37,10,21,20]. The weaker capabilities offered by the TPM makes them more suitable for static attestation than for a dynamic setting like the one we consider in this paper.

In recent independent work Pass, Shi and Tramer [35] formalize attestation guarantees offered by trusted hardware in the Universal Composability setting, and consider the feasibility of achieving UC-secure MPC from such assumptions. Interestingly, they show that in the setting that they consider (UC with a Global Setup (GUC) [12]) multiparty computation is impossible to achieve without additional assumptions, unless *all* parties have access to trusted hardware. They bypass this impossibility result by assuming

[7] `https://nacl.cr.yp.to`

that all parties have access to both trusted hardware as well some additional set-up. The resulting protocols are more intricate and less efficient than ours, so our results can be interpreted as a practice-oriented approach to the security of the most natural MPC protocol that relies on SGX, which trades composability for efficiency while still preserving strong privacy guarantees for the inputs to the computation. Furthermore, contrary to their approach, performing parallel executions of our protocol also entails several initializations, thus increasing performance overhead accordingly.

2 IEEs, Programs, and Machines

The models that we develop in this paper rely on the abstraction for IEEs introduced in [4]. Here we recall the key features of that model. A more in depth description of these formalisms is provided in the full version [3].

An IEE is viewed as an idealised machine running some fixed program P and which exposes an interface through which one can pass inputs and receive outputs to/from P. The I/O behaviour of a process running in an IEE is determined by the program it is running, and the inputs it receives. The interface models the strict isolation between processes running in different IEEs and formalizes that the only information that is revealed about a program running within an IEE is contained in its input-output behaviour.

PROGRAMS. We extend the model for programs from [4] to the setting where inputs/outputs are labeled: programs are transition functions which take a current state st and a label-input pair (l, i), and produce a new output o and an updated state. We write $o \leftarrow P[\mathsf{st}](l, i)$ for each such action and refer to it as an *activation*. Throughout the paper we restrict our attention to programs (even if they are adversarially created) for which the transition function is guaranteed to run in polynomial-time. Programs are assumed to be deterministic modulo of system calls; in particular we assume a system can call rand for providing programs with fresh randomness. Additionally, outputs are assumed to include a flag $\mathsf{finished}$ that indicating if the transition function will accept further input. We extend our notation to account for probabilistic programs that invoke the rand system call. We write $o \leftarrow P[\mathsf{st}; r](l, i)$ for the activation of P which when invoked on labeled input (l, i) (with internal state st and random coins r) produced output o. We write a sequence of activations as $(o_1, \ldots, o_n) \leftarrow P[\mathsf{st}; r](l_1, i_1, \ldots, l_n, i_n)$ and denote by $\mathsf{Trace}_{P[\mathsf{st};r]}(l_1, i_1, \ldots, l_n, i_n)$ the corresponding input/output trace $(l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$. For a trace T, we write $\mathsf{filter}[L](T)$ for the projection of the trace that retains only I/O pairs that correspond to labels in L. We use $\mathsf{filter}[l]$ when L is a singleton. We also extend the basic notion of program composition in [4] to consider label-based parallel and sequential program composition. Intuitively, when two labelled programs are composed, the set of labels of the composed program is enriched to encode the precise sub-program that should be activated and the label on which it should be activated.

MACHINES. As in [4] we model machines via a simple external interface, which we see as both the functionality that higher-level cryptographic schemes can rely on when using the machine, and the adversarial interface that will be the basis of our attack models. This interface can be thought of as an abstraction of Intel's SGX [27]. The

interface consists of three calls: 1. $\mathsf{Init}(1^\lambda)$ initialises the machine and outputs the global parameters prms. 2. $\mathsf{Load}(P)$ loads the program P in a fresh IEE and returns its handle hdl 3. $\mathsf{Run}(\mathsf{hdl}, l, i)$ passes the label-input pair (l, i) to the IEE with handle hdl. We define the I/O trace $\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$ of a process hdl running in a machine $\mathcal{M}$ as the tuple $(l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$ that includes the entire sequence of n inputs/outputs resulting from all invocations of Run on hdl; $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl})$ is the code (program) running under handle hdl; $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl})$ represents the coins given to the program by the rand system call; and $\mathsf{State}_{\mathcal{M}}(\mathsf{hdl})$ is the internal state of the program. Finally, we will write $\mathcal{A}^{\mathcal{M}}$ to indicate that algorithm $\mathcal{A}$ has access to machine $\mathcal{M}$.

3 Labelled Attested Computation

We now formalize a cryptographic primitive that generalizes the notion of Attested Computation proposed in [4], called Labelled Attested Computation. The main difference to the original proposal is that, rather than fixing a particular form of program composition for attestation, Labelled Attested Computation is agnostic of the program's internal structure; on the other hand, it permits controlling data flows and attestation guarantees via the label information included in program inputs.

SYNTAX. A *Labelled Attested Computation* (LAC) scheme is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, L^*)$ is the deterministic program compilation algorithm. On input global parameters for some machine $\mathcal{M}$, program P and an attested label set L^*, it outputs program P^*. This algorithm is run locally. P^* is the code to be run as an isolated process in the remote machine, whereas L^* defines which labelled inputs should be subject to attestation guarantees.
- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, l, i)$ is the stateless attestation algorithm. On input global parameters for $\mathcal{M}$, a process handle hdl and label-input pair (l, i), it uses the interface of $\mathcal{M}$ to obtain attested output o^*. This algorithm is run remotely, but in an unprotected environment: it is responsible for interacting with the isolated process running P^*, providing it with inputs and recovering the attested outputs that should be returned to the local machine.
- $\mathsf{Verify}(\mathsf{prms}, l, i, o^*, \mathsf{st})$ is the public (stateful) output verification algorithm. On input global parameters for $\mathcal{M}$, a label l, an input i, an attested output o^* and some state st it produces an output value o and an updated state, or the failure symbol $\perp$. This failure symbol is encoded so as to be distinguishable from a valid output of a program, resulting from a successful verification. This algorithm is run locally on claimed outputs from the Attest algorithm. The initial value of the verification state is set to be (prms, P, L^*), the same inputs provided to $\mathsf{Compile}$.

Intuitively, a LAC scheme is correct if, for any given program P and attested label set L^*, assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a partial view of the I/O sequence that took place in the remote environment, for an arbitrary set of labels L. A formal definition of correctness is provided in the full version [3].

Game $\mathsf{Att}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\!\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$; $(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\!\$\ \mathcal{A}_1(\mathsf{prms})$; $P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$; $\mathsf{st}_V \leftarrow (\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
  $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\!\$\ \mathcal{A}_2^\mathcal{M}(\mathsf{st}_\mathcal{A})$; $(o_k, \mathsf{st}_V) \leftarrow \mathsf{Verify}(\mathsf{prms}, l, i_k, o_k^*, \mathsf{st}_V)$
  If $o_k = \perp$ Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
  $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$; $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
  If $T \sqsubseteq T'$ Return F
Return T

Fig. 1. Game defining the security of LAC.

SECURITY. Security of labelled attested computation imposes that an adversary with control of the remote machine cannot convince the local user that some arbitrary remote (partial) execution of a program P has occurred, when it has not. It says nothing about the parts of the execution trace that are hidden from the client *or* are not in the attested label set L^*. Formally, we allow the adversary to freely interact with the remote machine, whilst providing a sequence of (potentially forged) attested outputs for a specific label $l \in L^*$. The adversary wins if the local user reconstructs an execution trace without aborting (i.e., all attested outputs must be accepted by the verification algorithm) and one of two conditions occur: i. there does not exist a remote process hdl^* running a compiled version of P where a consistent set of inputs was provided *for label* l; or ii. the outputs recovered by the local user for those inputs are not consistent with the semantics of P.

Technically, these conditions are checked in the definition by retrieving the full sequence of *label-input pairs* and random coins passed to all compiled copies of P running in the remote machine and running P on the same inputs to obtain the expected outputs. One then checks that for at least one of these executions, when the traces are restricted to special label l, that the expected trace matches the locally recovered trace via Verify. Since the adversary is free to interact with the remote machine as it pleases, we cannot hope to prevent it from providing arbitrary inputs to the remote program at arbitrary points in time, while refusing to deliver the resulting (possibly attested) outputs to the local user. This justifies the winning condition referring to a prefix of the execution in the remote machine, rather than imposing trace equality. Indeed, the definition's essence is to impose that, if the adversary delivers attested outputs for a particular label in the attested label set, then the subtrace of verified outputs for that label will be an exact prefix of the projection of the remote trace for that label.

We note that a higher-level protocol relying on LAC can fully control the semantics of labels, as these depend on the semantics of the compiled program. In particular, adopting the specific forms of parallel and sequential composition presented in Section 2, it is possible to use labels to get the attested execution of a sub-program that is fully isolated from other programs that it is composed with. This provides a much higher degree of flexibility than that offered by the original notion of Attested Computation.

Definition 1 (Security). *A labelled attested computation scheme is secure if, for all ppt adversaries $\mathcal{A}$, the probability that experiment in Fig. 1 returns T is negligible.*

The adversary loses the game if there exists at least one remote process that matches the locally reconstructed trace. This should be interpreted as the guarantee that IEE

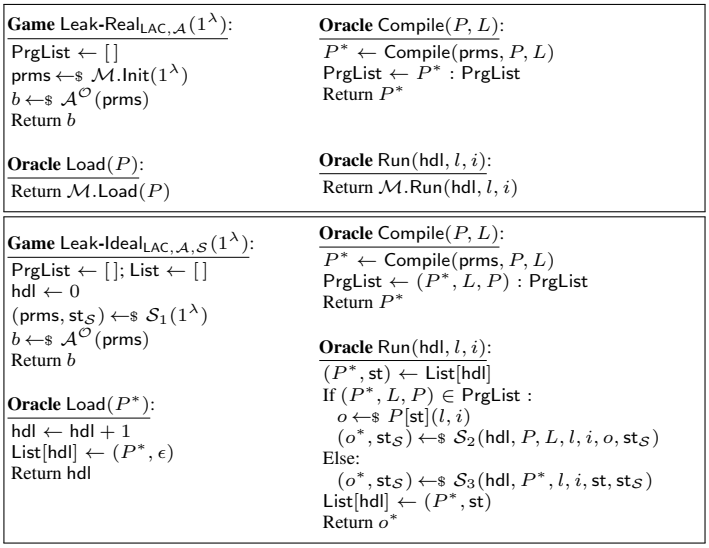

Fig. 2. Games defining minimum leakage of LAC.

resources are indeed being allocated in a specific remote machine to run at least one instance of the remote program (note that if the program is deterministic, many instances could exist with exactly the same I/O behaviour, which is *not* seen as a legitimate attack). Furthermore, our definition imposes that the compiled program uses essentially the same randomness as the source program (except of course for randomness that the security module internally uses to provide its cryptographic functionality), as otherwise it is easy for the adversary to make the (idealized) local trace diverge from the remote. This is a consequence of our modelling approach, but it does not limit the applicability of our primitive: it simply spells out that the transformation performed on the code for attestation will typically consist of an instrumentation of the code by applying cryptographic processing to the inputs and outputs it receives.

MINIMAL LEAKAGE. The above discussion shows that a LAC scheme guarantees that the I/O behaviour of the program in the remote machine includes at least the information required to reconstruct an hypothetical local execution of the source program. Next, we require that a compiled program does not reveal any information beyond what the original program would reveal. The following definition imposes that nothing from the internal state of the source programs (in addition to what is public, i.e., the code and I/O sequence) is leaked in the trace of the compiled program.

Definition 2 (Minimal leakage). *A labelled attested computation scheme* LAC *ensures security with minimal leakage if it is secure according to Definition 1 and there exists a ppt simulator $\mathcal{S}$ that, for every adversary $\mathcal{A}$, the following distributions are identical:*

$$\{\, \mathsf{Leak\text{-}Real}_{\mathsf{LAC},\mathcal{A}}(1^\lambda) \,\} \;\approx\; \{\, \mathsf{Leak\text{-}Ideal}_{\mathsf{LAC},\mathcal{A},\mathcal{S}}(1^\lambda) \,\}$$

where games Leak-Real$_{\mathsf{LAC},\mathcal{A}}$ *and* Leak-Ideal$_{\mathsf{LAC},\mathcal{A},\mathcal{S}}$ *are shown in Fig. 2.*

Intuitively, this means that one can construct a perfect simulation of the remote trace by simply appending cryptographic material to the local trace. This property is important when claiming that the security of a cryptographic primitive is preserved when it is run within an attested computation scheme.

4 LAC from SGX-like systems

Our labelled attested computation protocol relies on the capabilities offered by the security module of Secure Guard Extensions (SGX) architecture proposed by Intel [2] (i.e. MACs for authenticated communication between IEEs, and digital signatures for inter-platform attestation of executions). Our security module formalization is the same as the one adopted in [4].

SECURITY MODULE. The security module relies on a signature scheme and a MAC scheme,and operates as follows:

- On initialization, the security module generates a key pair $(\mathsf{pk}, \mathsf{sk})$ and a symmetric key key.It also creates a special process running code S^* in an IEE with handle 0. The security module then securely stores the key material, and outputs the public key.
- The operation of IEE with handle 0 is different from all other IEEs in the machine. Program S^* will permanently reside in this IEE, and it will be the only one with direct access to both sk and key. The code of S^* is dedicated to transforming messages authenticated with key into messages signed with sk. On activation, it expects an input (m, t). It obtains key from the security module and verifies the tag. If the previous operation was successful, it obtains sk from the security module, signs the message and outputs the signature.
- The security module exposes a single system call $\mathsf{mac}(\mathsf{m})$ to code running in all other IEEs. On such a request from a process running program P, the security module returns a MAC tag t computed using key over both the code of P and the input message m.

LABELLED ATTESTED COMPUTATION SCHEME. We now define a LAC scheme that relies on a remote machine supporting such a security module. Basic replay protection using a sequence number does not suffice to bind a remote process to a subtrace, since the adversary could then run multiple copies of the same process and *mix and match* outputs from various traces. This is similar to the reasoning in [4]. However, in this paper we are interested in validating traces for specific attested labels, independently from each other, rather than the full remote trace. Our LAC scheme works as follows:

- $\mathsf{Compile}(\mathsf{prms}, P, L)$ generates a new program P^* and outputs it. Program P^* is instrumented as follows:
  - in addition to the internal state st of P, it maintains a list ios_l of all the I/O pairs it has previously received and computed for each label $l \in L$.
  - On input (l, i), P^* computes $o \leftarrow\!\!\$\ P[\mathsf{st}_P](l, i)$ and verifies if $l \in L$. If this is not the case, then P^* simply outputs non-attested output o.
  - Otherwise, it updates the list ios by appending (l, i, o), computes the subset of ios for label $l : \mathsf{ios}_l \leftarrow \mathsf{filter}[l](\mathsf{ios})$ and requests from the security module a MAC of for that list. Due to the operation of the security module, this will correspond to a tag t on the tuple (P^*, ios_l).

- It finally outputs $(o, \mathsf{t}, P^*, \mathsf{ios}_l)$. We note that we include (P^*, ios_l) explicitly in the outputs of P^* for clarity of presentation only. This value would be kept in an insecure environment by a stateful Attest program.

– $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, l, i)$ invokes $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, (l, i))$ using the handle and input value it has received. Attest then checks is the produced output o is to be attested and if so transforms the tag into a signature σ using the IEE with handle 0 and outputs (o', σ). Otherwise it simply outputs o.

– $\mathsf{Verify}(\mathsf{prms}, l, i, o^*, \mathsf{st})$ is the stateful verification algorithm. Initially $\mathsf{st} = (\mathsf{prms}, P, L^*)$, on first activation Verify computes and stores P^* and initialises an empty list ios of input-output pairs. Verify returns o^* if $l \notin L$. Otherwise, it first parses o^* into (o, σ), appends (l, i, o) to ios and verifies the digital signature σ using prms and $(P^*, \mathsf{filter}[l](\mathsf{ios}))$. If parsing or verification fails, Verify outputs $\perp$. If not, then Verify outputs o.

Correctness of our LAC scheme is clear – a detailed analysis is in the full version [3].

Theorem 1 (LAC scheme security). *The LAC scheme presented above provides secure attestation if the underlying MAC scheme* Π *and signature scheme* Σ *are existentially unforgeable. Furthermore, it unconditionally ensures minimum leakage.*

The proof of this theorem generalizes that of basic attestation schemes in [4] and can be found in the full version [3]. All attested outputs are bound to a partial execution trace that contains the entire I/O sequence associated with the corresponding attested label, so all messages accepted by Verify must exist as a prefix for a remote trace of some instance of P^*. The adversary can only cause an inconsistency in $T \sqsubseteq T'$ if the signature verification performed by Verify accepts a message of label $l \in L^*$ that was never authenticated by an IEE running P^*. However, in this case the adversary either breaks the MAC scheme (and dishonestly executing Attest), or breaks the signature (directly forging attested outputs).

5 Secure computation with IEEs

FUNCTIONALITIES. We want to securely execute a functionality $\mathcal{F}$ defined by a four-tuple $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout})$, where F is a deterministic stateful transition function that takes inputs of the form (id, i). Here, id is a party identifier, which we assume to be an integer in the range $[1..n]$, and n is the total number of participating parties. On each transition, F produces an output that is intended for party id, as well as an updated state. We associate to F two leakage functions $\mathsf{Lin}(k, i, \mathsf{st})$ and $\mathsf{Lout}(k, o, \mathsf{st})$ which define the public leakage that can be revealed by a protocol about a given input i or output o for party k, respectively; for the sake of generality, both functions may depend on the internal state st of the functionality, although this is not the case in the examples we consider in this paper. Arbitrary reactive functionalities formalized in the Universal Composability framework can be easily recast as a transition function such as this. The upside of our approach is that one obtains a precise code-based definition of what the functionality should do (this is central to our work since these descriptions give rise to concrete programs); the downside is that the code-based definitions may be less clear to a human reader, as one cannot ignore the tedious *book-keeping* parts of the functionality.

EXECUTION MODEL. We assume the existence of a machine $\mathcal{M}$ allowing for the usage of isolated execution environments, as defined in Section 2. In secure computation terms, this machine should *not* be seen as an ideal functionality that enables some hybrid model of computation, but rather an additional party that comes with a specific setup assumption, a fixed internal operation, and which cannot be corrupted. Importantly, all interactions with $\mathcal{M}$ and all the code that is run in $\mathcal{M}$ but outside IEEs is considered to be adversarially controlled.

SYNTAX. A protocol π for functionality $\mathcal{F}$ is a seven-tuple of algorithms as follows:

- Setup – This is the party local set-up algorithm. Given the security parameter, the public parameters prms for machine $\mathcal{M}$ and the party's identifier id, it returns the party's initial state st (incluing its secret key material) and its public information pub.
- $\mathsf{Compile}$ – This is the (deterministic) code generation algorithm. Given the description of a functionality F, and public parameters $(\mathsf{prms}, \mathsf{Pub})$ for both the remote machine and the entire set of public parameters for the participating parties, it generates the instrumented program that will run inside an IEE.
- Remote – This is the untrusted code that will be run in $\mathcal{M}$ and which ensures the correctness of the protocol by controlling its scheduling and input collection order. It has oracle access to $\mathcal{M}$ and is in charge of collecting inputs and delivering outputs. Its initial state describes the order in which inputs of different parties should be provided to the functionality.
- Init – This is the party local protocol initialization algorithm. Given the party's state st produced by Setup and the public information of all participants Pub it outputs an uptated state st. We note that a party can choose to engage in a protocol by checking if the public parameters of all parties are correct and assigned to roles in the protocol that match the corresponding identities.
- $\mathsf{AddInput}$ – This is the party local input providing algorithm. Given the party's current state st and an input in, it outputs an uptated state st.
- $\mathsf{Process}$ – This is the party local message processing algorithm. Given its internal state st, and an input message m, it runs the next protocol stage, updates the internal state and returns output message m'.
- Output – This is the party local output retrieval algorithm. Given internal state st, it returns the current output o.

Intuitively such a protocol is correct if it can support any execution schedule whilst evaluating the functionality correctly. A precise definition is provided in the full version [3].

SECURITY. As is customary in secure computation models, we take the ideal world versus real world approach to define security of a protocol. Our security model is presented in Figure 3, and is described as follows. In the real world, the adversary interacts with an IEE-enabled machine $\mathcal{M}$ under adversarial control and oracles $\mathsf{SetInput}$, $\mathsf{GetOutput}$ and Send providing it with the locally run part of the protocol. In the ideal world, the adversary is presented with 1. a simulator $\mathcal{S}$ emulating the remote machine, the setup phase, and the Send oracle 2. idealised oracles $\mathsf{SetInput}$, $\mathsf{GetOutput}$. The idealised oracles only do book-keeping of which input should be transmitted next and which output should be retrieved next for each honest party. $\mathcal{S}$ gets given oracle access to a

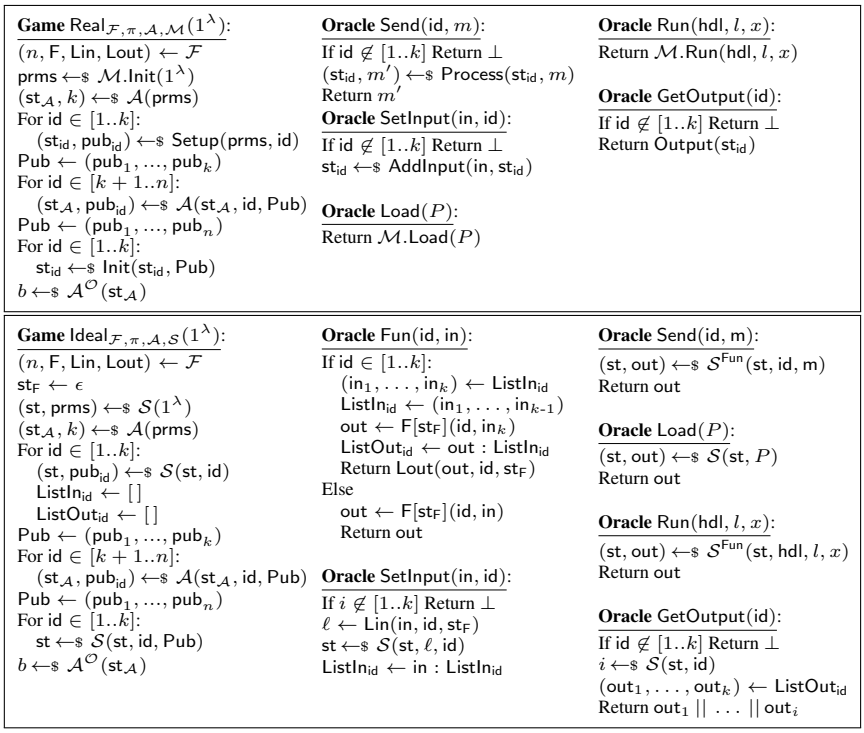

Fig. 3. Real and Ideal security games.

functionality evaluation oracle Fun that consumes the next input of a party (defined in SetInput if the party is honest, passed as input otherwise) and sets the next output for this party, returning the leakage of the input and output if the party is honest, and the full output otherwise. A protocol is deemed secure if there exists a ppt simulator such that no ppt adersary can distinguish the two worlds.

Definition 3. *We say π is secure for $\mathcal{F}$ if, for any ppt adversary $\mathcal{A}$, there exists a ppt simulator $\mathcal{S}$ such that the following definition of advantage is a negligible function in the security parameter.*

$$|\,\mathsf{Pr}[\mathsf{Real}^{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow b = 1] - \mathsf{Pr}[\mathsf{Ideal}^{\mathcal{F},\pi,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow b = 1]\,|$$

Succinctly, our model is inspired in the UC framework, and can be derived from it when natural restrictions are imposed: PKI, static corruptions, and a distinguished non-corruptible party modeling an SGX-enabled machine.[8] A security proof for a protocol in our model can be interpreted as translation of any attack against the protocol in the real world, as an attack against the ideal functionality in the ideal world. The simulator performs this translation by presenting an execution environment to the adversary that is consistent with what it is expecting in the real world. It does this by simulating the

[8] This particular choice in our model has implications for the composability properties of our results, as discussed in the related work section.

operations of the Load, Run and Send oracles, which represent the operation of honest parties in the protocol. While the adversary is able to provide the inputs and read the outputs for honest parties directly from the functionality, the simulator is only able obtain partial leakage about this values via the Lin and Lout functions. Conversely, it can obtain the functionality outputs for corrupt parties via the Fun oracle and, furthermore, it is also able to control the rate and order in which all inputs are provided to the functionality. Were this not the case, the adversary would be able to distinguish the two worlds by manipulating scheduling in a way the simulator could not possibly match.

6 A New MPC Protocol from SGX

In this section we describe our secure multiparty computation protocol based on LAC that works for any functionality. The protocol starts by running bootstrap code in an isolated execution environment in the remote machine; the code exchanges keys with each of the participants in the protocol. These key exchange programs are composed in parallel, as seen in Section 2. We reuse the notion of AttKE (key exchange for attested computation) from [4] which provides the right notion of key exchange security in this context. Once this bootstrap stage is concluded, the code of the functionality starts executing. The functionality uses the secure channels established before to ensure that the inputs and outputs are private and authenticated. The security of this protocol relies on a utility theorem similar to that of [4] for the use of key exchange in the context of attestation.

Program $\mathsf{Box}\langle\mathcal{F}, \Lambda\rangle[\mathsf{st}](i^*, l)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{id} \leftarrow l$
If $\mathsf{id} \notin [1..n]$: Return $\perp$
If $\mathsf{st.seq_{id}} = \epsilon$: $\mathsf{st.seq_{id}} \leftarrow 0$
$i \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st.key_{id}}, \mathsf{m})$
If $\mathsf{m} \neq (\mathsf{in}, \mathsf{st.seq_{id}})$: Return $\perp$
$o \leftarrow F[\mathsf{st.st_F}](\mathsf{id}, \mathsf{in})$
$\mathsf{st.seq_{id}} \leftarrow \mathsf{st.seq_{id}} + 1$
$c \leftarrow\!\!\$\ \Lambda.\mathsf{Enc}(\mathsf{st.key_{id}}, (\mathsf{seq}, o))$
$\mathsf{st.seq_{id}} \leftarrow \mathsf{st.seq_{id}} + 1$
Return c

Fig. 4. Boxing using Authenticated Encryption

Theorem 2 (Local AttKE utility). *If the AttKE is correct and secure, and the LAC protocol is correct, secure and ensures minimal leakage, then for all ppt adversaries in the labelled utility experiment: the probability that the adversary violates the AttKE two-sided entity authentication is negligible; and the key secrecy advantage* $2 \cdot \Pr[\mathsf{guess}] - 1$ *is negligible.*

This theorem shows that, under the specific program composition pattern that we require for our MPC protocol, which guarantees AttKE isolation from other programs, each party obtains a secret key that is indistinguishable from a random string. The detailed labelled utility experiment and the theorem proof can be found in the full version [3]. It follows that the key can be used to construct a secure channel that connects it to code emulating the functionality within an IEE.

BOXING USING AUTHENTICATED ENCRYPTION. As explained above, after the bootstrapping stage of our protocol, we run the ideal functionality within an isolated execution environment. We implement this part of the execution using the *boxing* construction shown in Figure 4. The name comes by analogy with placing the functionality within a box, which parties can access via secure channels. The labelled program $\mathsf{Box}\langle\mathcal{F}, \Lambda\rangle$ is

parametrized by a functionality $\mathcal{F}$ for n parties and a secure authenticated encryption encryption scheme Λ. Its initial state is assumed to contain n symmetric keys compatible with Λ, denoted sk_1 to sk_n (one for each participating party) and the empty initial state for the functionality st_F. The Box expects encrypted inputs i^* under a label l identifying the party providing the input. These are then decrypted using the respective key sk_l and provided to $\mathcal{F}$. The value returned by the functionality is encrypted using the same sk_l and is then returned. To avoid replays of encrypted messages, we keep one sequence number seq_id per communicating party id.

THE PROTOCOL. Building on top of a LAC scheme, an AttKE scheme and our Box construction we define a general secure multiparty computation protocol that works for any (possibly reactive) functionality F. The core of the protocol is the execution of an AttKE for each participant in parallel, followed by the execution of the functionality F on the remote machine, under a secure channel with each participant as specified in the Box construct. More precisely:

- Setup derives the code for a remote key exchange program Rem_KE using the AttKE setup procedure. This code (which intuitively includes cryptographic public key material) is set to be the public information for this party. The algorithm also stores various parameters in the local state for future usage.
- $\mathsf{Compile}$ uses the LAC compilation algorithm on a program that results from the parallel composition of all the remote key exchange programs for all parties, which is then sequentially composed with the boxed functionality.
- Init locally reconstructs the program that is intended for remote execution, as this is needed for attestation verification. The set of labels that define the locally recovered trace is set to the pair $((p, (\mathsf{id}, \epsilon)), (q, \mathsf{id}))$, corresponding to the parts of the remote trace that are relevant for this party, namely its key exchange and its inputs/outputs.
- $\mathsf{Process}$ is split into two stages. In the first stage it uses LAC with attested labels of the form $(p, (\mathsf{id}, \epsilon))$ to execute AttKE protocol and establish a secure channel with the remote program. In the second stage, it uses non-attested labels of the form (q, id), and it provides inputs to the remote functionality (on request) and recovers the corresponding outputs when they are delivered.
- Output reads the output in the state of the participant and returns it.
- $\mathsf{AddInput}$ adds an input to the list of inputs to be transmitted by the participant.

Pseudo code of the protocol as well as formal details of the (untrusted) scheduling algorithm can be found in the full version [3].

For proving security, we restrict the functionalities we consider to a particular leakage function: size of inputs/outputs. We say that a functionality $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout})$ leaks size if it is such that Lin and Lout return the length of the inputs/outputs ($\mathsf{Lin}(k, x, \mathsf{st}) = \mathsf{Lout}(k, x, \mathsf{st}) = |x|$ for every k, x, st).

Theorem 3. *If* LAC *is a correct and secure LAC scheme,* AttKE *is a secure AttKE scheme and Λ a secure authenticated encryption scheme, then the protocol described in this Section is correct and secure for any functionality that leaks size.*

PROOF SKETCH. We build the required simulator $\mathcal{S}$ as follows. For dishonest parties, the simulator executes the protocol normally while for the honest parties instead of

encrypting the inputs/outputs the simulator encrypts dummy messages of the correct length (obtained through the leakage function) under freshly generated keys.

We sketch a proof of indistinguishability between the real and ideal worlds. The full proof can be found in the full version [3]. The proof consists of 3 hops, the first is a hybrid argument over the honest parties. In this hybrid argument one gradually replaces the key derived by each honest party by a random one. In each step, the utility theorem is used to show that this change cannot be noticed by the adversary. In the second hop, we replace the encrypted inputs/outputs for honest parties by encrypted dummy payloads of the correct length. This hop is correct by the indistinguishability of encrypted ciphertexts. After this game hop, the resulting game is *identical until bad* to the ideal world, where the bad event corresponds to the simulator aborting due to an inconsistent message being accepted as the next undelivered input or output. Due to the use of sequence numbers, this bad event can be reduced to the authenticity of the encryption scheme.

7 Implementation

We provide an implementation of our protocol sgx-mpc-nacl relying on NaCl for the cryptographic library and Intel SGX for the IEEs. We use elliptic curves for both the key exchange (Diffie-Hellman) and signatures, and a combination of the Salsa20 and Poly1305 encryption and authentication schemes [8] for authenticated encryption. Our implementation relies on Intel's Software Development Kit for dealing with the SGX low-level operations. These include loading code into an IEE (our Load abstraction), calling a function within the IEE (our Run abstraction), and constructing an attested message (first getting a MAC'ed message within the IEE, and using the quoting enclave to convert it into a signature). It employs the LAC scheme proposed here, and include wrappers that match our abstractions of signatures and authenticated encryption. These are then used to construct the bootstrapping protocol (AttKE) that enables each party to establish an independent secret key and secure channel to communicate with the Box construction running inside the IEE. Finally, our implementation of the Box is agnostic of the intended functionality, and can be linked to arbitrary functionality implementations, provided that these comply with a simple labelled I/O interface. The top-level interface to our protocol includes the code that runs inside the IEE, the code that runs outside the IEE in the remote machine for book-keeping operations and the client-side code that bootstraps a secure channel and then sends/receives messages from the functionality.

We compare our implementation with measurements we performed using the ABY framework [19]. We chose ABY for comparison, as we could evaluate it on the same platform we used for assessing our protocol, therefore avoiding differences due to performance disparities of heterogeneous evaluation platforms. Although it is specific to the two-party secure computation setting, ABY is representative of state-of-the-art MPC implementations and we expect results for other frameworks such as Sharemind [16] and SPDZ [18] to lead to similar conclusions; indeed, the crux of our performance gains resides in the fact that our solution does not require encoding the computation in circuit form, unlike all the aforementioned protocols.[9]

[9] We also note that ABY assumes a semi-honest adversary, which is weaker than the one we consider; but still our performance gains are significant.

	Phase	Preparation (ms)		Online (ms)		Total (ms)	
	Protocol	**ABY**	**Ours**	**ABY**	**Ours**	**ABY**	**Ours**
Input size (bits)	*160*	196.3	115.7	0.752	0.050	197.1	117.75
	1,600	196.7	115.7	1.819	0.302	198.5	116.00
	16,000	201.6	115.7	13.14	2.798	214.7	118.50
	160,000	226.2	115.2	144.4	27.77	370.6	142.97

Phase	**ABY**	**Ours**
Preparation (ms)	197.9	115.84
Online (ms)	3.249	0.661
Total (ms)	201.1	116.50

	Phase	Preparation		Online		Total	
	Protocol	**ABY**	**Ours**	**ABY**	**Ours**	**ABY**	**Ours**
Set size	*100*	224.8	115.8	1.084	0.043	225.9	115.84
	1000	368.1	115.8	2.168	0.199	370.3	116.00
	10,000	1442.2	115.8	12.88	1.758	1455.1	117.56
	100,000	10,698.7	115.7	109.5	17.39	10,808.2	133.09
	1,000,000	84,096.6	115.7	1616.0	173.1	85,712.6	288.80

Phase	**ABY**	**Ours**
Preparation (ms)	196.3	127.7
Online (ms)	0.404	0.024
Total (ms)	196.7	127.7

Table 1. Clockwisely, starting from upper left: hamming distance, AES, millionaire's problem and private set intersection

Like our protocol, the ABY protocol has two phases: a *preparation phase* and an *online phase*. The preparation phase comprises the key exchange between the input parties by means of oblivious transfer (OT), and generation of the garbled circuit (GC) representing the desired function. In the online phase the GC gets evaluated and the result is sent back to the output party. In our protocol, the preparation phase is used to establish a secure channel between the IEE and the input parties. The online phase of our protocol comprises the decryption of inputs in the Box component, the evaluation of the payload function, and the encryption of the results, again by the Box component.

We evaluated the performance of four different secure two-party computation use cases (Table 1): AES, millionaire's problem, private set intersection and hamming distance. In comparison to ABY, the preparation phase and online phase are shorter with sgx-mpc-nacl, and consequently the overall runtime is faster as well. In general, sgx-mpc-nacl is faster for all the testing computations performed. However, the gains are considerably more noticeable when we increase with input size and computation. This has the highest significance on evaluation of the private set intersection with the largest input size (1 mill.), where our implementation is roughly 300 times faster.

SIDE CHANNELS AND SOFTWARE RESILIENT AGAINST TIMING ATTACKS. Recent works [38,15] have pointed out that IEE-enabled systems such as Intel's SGX do not offer more protection against side-channel attacks than traditional microprocessors. This is a relevant concern, since the IEE trust model which we also adopt in this paper admits that the code outside IEEs is potentially malicious and that the machine is under the control of an untrusted party. We believe that there are two aspects to this problem that should be considered separately. The first aspect is the production of the IEE-enabled hardware/firmware itself and the protection of the long-term secrets used by the attestation security module. If the computations performed by the attestation infrastructure itself are vulnerable, then there is nothing that can be done at the protocol design/implementation level. This aspect of trust is within the remit of the manufacturers.

An orthogonal issue is the possibility that software running inside an IEE leaks part of its state or short-term secrets via side channels. One should distinguish between software observations and hardware/physical observations. In the former, software co-located in the machine observes timing channels based on memory access patterns, control flow, branch prediction, cache-based based attacks [15], page-fault side channels [38], etc. Protection against these side-channel attacks has been widely studied in the practical

crypto community, where a consensus exists that writing so-called *constant-time* software is the most effective countermeasure [7,30]. As mentioned above, constant-time software has the property that the entire sequence of memory addresses (in both data and code memory) accessed by a program can be predicted in advance from public inputs, e.g., the length of messages. When it comes to hardware/physical side-channel attacks such as those relying on temperature measurements, power analysis, or electromagnetic radiation, the effectiveness of software countermeasures is very limited, and improving hardware defenses again implies obtaining additional guarantees from the equipment manufacturer.

Our implementation sgx-mpc-nacl enforces a strict constant-time policy that is consistent with the IEE trust model. To provide a protocol that is fully constant-time, one must also ensure that the executed functionality is constant-time. Recent work in the formal verification area sheds new light how this can be achieved over low-level code in a fully automatic way [1].

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*. USENIX Association, 2016.
2. I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.
3. R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A. Sadeghi, G. Scerri, and B. Warinschi. Secure multiparty computation from SGX. *IACR Cryptology ePrint Archive*, 2016.
4. M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to SGX. In *EuroS&P*. IEEE, 2016.
5. A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*. USENIX Association, 2014.
6. A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS*. ACM, 2008.
7. D. J. Bernstein. Cache-timing attacks on aes, 2005. `cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
8. D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, volume 7533 of *LNCS*. Springer, 2012.
9. P. Bogetoft, I. Damgård, T. P. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography*, volume 4107 of *LNCS*. Springer, 2006.
10. E. Brickell, L. Chen, and J. Li. A new direct anonymous attestation scheme from bilinear maps. In *TRUST*, volume 4968 of *LNCS*. Springer, 2008.
11. E. F. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS*. ACM, 2004.
12. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC*, volume 4392 of *LNCS*. Springer, 2007.
13. R. Canetti and M. Fischlin. Universally composable commitments. In *CRYPTO*, volume 2139 of *LNCS*. Springer, 2001.
14. L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted virtual domains - design, implementation and lessons learned. In *INTRUST*, volume 6163 of *LNCS*. Springer, 2009.
15. V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016.
16. CYBERNETICA. Sharemind. `sharemind.cyber.ee/`.

17. I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft. Confidential benchmarking based on multiparty computation. *IACR Cryptology ePrint Archive*, 2015.
18. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *LNCS*. Springer, 2012.
19. D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.
20. A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *DATE*. European Design and Automation Association, 2014.
21. H. Ge and S. R. Tate. A direct anonymous attestation scheme for embedded devices. In *PKC*, volume 4450 of *LNCS*. Springer, 2007.
22. C. Gebhardt and A. Tomlinson. Secure virtual disk images for grid computing. In *APTC*. IEEE, 2008.
23. D. Gupta, B. Mood, J. Feigenbaum, K. R. B. Butler, and P. Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *FC Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, volume 9604 of *LNCS*. Springer, 2016.
24. S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *CRYPTO*, volume 6841 of *LNCS*. Springer, 2011.
25. W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*. ACM, 2010.
26. M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ISCA*. ACM, 2013.
27. Intel. *Software Guard Extensions Programming Reference*, 2014. `software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.
28. J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, volume 4515 of *LNCS*. Springer, 2007.
29. P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *EuroSys*. ACM, 2014.
30. A. Langley. Lucky thirteen attack on TLS CBC. `www.imperialviolet.org/2013/02/04/luckythirteen.html`, 2013.
31. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*. USENIX, 2004.
32. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *EuroSys*. ACM, 2008.
33. Microsoft. *BitLocker Drive Encryption: Data Encryption Toolkit for Mobile PCs: Security Analysis*, 2007. `technet.microsoft.com/en-us/library/cc162804.aspx`.
34. J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*. USENIX Association, 2013.
35. R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. *IACR Cryptology ePrint Archive*, 2016.
36. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *S&P*. IEEE, 2015.
37. B. Smyth, M. Ryan, and L. Chen. Direct anonymous attestation (DAA): ensuring privacy with corrupt administrators. In *ESAS*, volume 4572 of *LNCS*. Springer, 2007.
38. Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*. IEEE, 2015.