Formal Abstractions for Attested Execution Secure Processors

Rafael Pass
CornellTech

Elaine Shi
Cornell

Florian Tramèr
Stanford

Abstract

Realistic secure processors, including those built for academic and commercial purposes, commonly realize an "attested execution" abstraction. Despite being the *de facto* standard for modern secure processors, the "attested execution" abstraction has not received adequate formal treatment. We provide formal abstractions for "attested execution" secure processors and rigorously explore its expressive power. Our explorations show both the expected and the surprising.

On one hand, we show that just like the common belief, attested execution is extremely powerful, and allows one to realize powerful cryptographic abstractions such as stateful obfuscation whose existence is otherwise impossible even when assuming virtual blackbox obfuscation and *stateless* hardware tokens. On the other hand, we show that surprisingly, realizing *composable* two-party computation with attested execution processors is not as straightforward as one might anticipate. Specifically, only when both parties are equipped with a secure processor can we realize composable two-party computation. If one of the parties does not have a secure processor, we show that composable two-party computation is impossible. In practice, however, it would be desirable to allow multiple legacy clients (without secure processors) to leverage a server's secure processor to perform a multi-party computation task. We show how to introduce minimal additional setup assumptions to enable this. Finally, we show that *fair* multi-party computation for general functionalities is impossible if secure processors do not have trusted clocks. When secure processors have trusted clocks, we can realize fair two-party computation if both parties are equipped with a secure processor; but if only one party has a secure processor (with a trusted clock), then fairness is still impossible for general functionalities.

Contents

1 Introduction

The science of cybersecurity is founded atop one fundamental guiding principle, that is, to minimize a system's Trusted Computing Base (TCB) [78]. Since it is notoriously difficult to have "perfect" software in practice especially in the presence of legacy systems, the architecture community have advocated a new paradigm to bootstrap a system's security from trusted hardware (henceforth also referred to as secure processors). Roughly speaking, secure processors aim to reduce a sensitive application's trusted computing base to only the processor itself (possibly in conjunction with a minimal software TCB such as a secure hypervisor). In particular, besides itself, a sensitive application (e.g., a banking application) should not have to trust any other software stack (including the operating system, drivers, and other applications) to maintain the confidentiality and/or integrity of mission-critical data (e.g., passwords or credit card numbers). Security is retained even if the software stack can be compromised (as long as the sensitive application itself is intact). Besides a software adversary, some secure processors make it a goal to defend against physical attackers as well. In particular, even if the adversary (e.g., a rogue employee of a cloud service provider or a system administrator) has physical access to the computing platform and may be able to snoop or tamper with memory or system buses, he should not be able to harvest secret information or corrupt a program's execution.

Trusted hardware is commonly believed to provide a very powerful abstraction for building secure systems. Potential applications are numerous, ranging from cloud computing [13, 35, 59, 69, 70], mobile security [68], web security, to cryptocurrencies [82]. In the past three decades, numerous secure processors have been proposed and demonstrated by both academia and industry [7, 28, 34, 39, 40, 57, 58, 60, 74, 81]; and several have been commercialized, including the well-known Trusted Platform Modules (TPMs) [2], Arm's TrustZone [6, 8], and others. Notably, Intel's recent release of its new x86 security extensions called SGX [7, 33, 60] has stirred wide-spread interest to build new, bullet-proof systems that leverage emerging trusted hardware offerings.

1.1 Attested Execution Secure Processors

Although there have been numerous proposals for the design of trusted hardware, and these designs vary vastly in terms of architectural choices, instruction sets, implementation details, cryptographic suites, as well as adversarial models they promise to defend against — amazingly, it appears that somehow most of these processors have converged on providing a common abstraction, henceforth referred to as the *attested execution* abstraction [2, 7, 34, 60, 74, 77]. Roughly speaking, an attested execution abstraction enables the following:

- A platform equipped with an attested execution processor can send a program and inputs henceforth denoted $(\mathsf{prog}, \mathsf{inp})$ to its local secure processor. The secure processor will execute the program over the inputs, and compute $\mathsf{outp} := \mathsf{prog}(\mathsf{inp})$. The secure processor will then sign the tuple $(\mathsf{prog}, \mathsf{outp})$ with a secret signing key to obtain a digital signature σ — in practice, a hash function is applied prior to the signing. Particularly, this signature σ is commonly referred to as an "attestation", and therefore this entire execution is referred to as an "attested execution".

- The execution of the aforementioned program is conducted in a sandboxed environment (henceforth referred to as an *enclave*), in the sense that a software adversary and/or a physical adversary cannot tamper with the execution, or inspect data that lives inside the enclave. This is important for realizing privacy-preserving applications. For example, a remote client who knows the

secure processor's public key can establish a secure channel with a secure processor residing on a remote server $\mathcal{S}$. The client can then send encrypted and authenticated data (and/or program) to the secure processor — while the messages are passed through the intermediary $\mathcal{S}$, $\mathcal{S}$ cannot eavesdrop on the contents, nor can it tamper with the communication.

- Finally, various secure processors make different concrete choices in terms of how they realize such secure sandboxing mechanisms as mentioned above — and the choices are closely related to the adversarial capabilities that the secure processor seeks to protect against. For example, roughly speaking, Intel's SGX technology [7, 60] defends against a restricted software adversary that does not measure timing or other possible side channels, and does not observe the page-swap behavior of the enclave application (e.g., the enclave application uses small memory or is by design data-oblivious); it also defends against a restricted physical attacker capable of tapping memory, but not capable of tapping the addresses on the memory bus or measuring side-channel information such as timing and power consumption.

  We refer the reader to Shi et al. [72] for a general-purpose introduction of trusted hardware, and for a comprehensive comparison of the different choices made by various secure processors.

The fact that the architecture community has converged on the "attested execution" abstraction is intriguing. How exactly this has become the *de facto* abstraction is beyond the scope of this paper, but it is helpful to observe that the attested execution abstraction is cost-effective in practice in the following senses:

- *General-purpose:* The attested execution abstraction supports the computation of general-purpose, user-defined programs inside the secure enclave, and therefore can enable a broad range of applications;
- *Reusability:* It allows a single trusted hardware token to be reused by multiple applications, and by everyone in the world — interestingly, it turns out such reusability actually gives rise to many of the technicalities that will be discussed later in the paper;
- *Integrity and privacy:* It offers both integrity and privacy guarantees. In particular, although the platform $\mathcal{P}$ that is equipped with the trusted hardware serves an intermediary in every interaction with the trusted hardware, privacy guarantees can be bootstrapped by having remote users establish a secure channel with the secure processor.

In the remainder of the paper, whenever we use the term "secure processors" or "trusted hardware", unless otherwise noted we specifically mean attested execution secure processors.

1.2 Why Formal Abstractions for Secure Processors?

Although attested execution has been accepted by the community as a *de facto* standard, to the best of our knowledge, no one has explored the following fundamental questions:

1. Precisely and formally, what is the attested execution abstraction?
2. What can attested execution express and and what can it not express?

If we can formally and precisely articulate the answers to these questions, the benefits can be wide-spread. It can help both the producer as well as the consumer of trusted hardware, in at least the following ways:

- *Understand whether variations in abstraction lead to differences in expressive power.* First, various secure processors may provide similar but subtly different abstractions — do these differences matter to the expressive power of the trusted hardware? If we wish to add a specific feature to a secure processor (say, timing), will this feature increase its expressive power?

- *Enable formally correct use of trusted hardware.* Numerous works have demonstrated how to use trusted hardware to build a variety of secure systems [13, 14, 29, 35, 59, 64, 67, 69–71]. Unfortunately, since it is not even clear what precise abstraction the trusted hardware offers, the methodology adopted by most existing works ranges from heuristic security to semi-formal reasoning.

  Moreover, most known secure processors expose cryptography-related instructions (e.g., involving hash chains or digital signatures [2,7,33,60]), and this confounds the programming of trusted hardware — in particular, the programmer essentially has to design cryptographic protocols to make use of trusted hardware. It is clear that user-friendly higher-level programming abstractions that hide away the cryptographic details will be highly desirable, and may well be the key to the democratization of trusted hardware programming (and in fact, to security engineering in general) — and yet without precisely articulating the formal abstraction trusted hardware offers, it would clearly be impossible to build formally correct higher-level programming abstractions atop.

- *Towards formally secure trusted hardware.* Finally, understanding what is a "good" abstraction for trusted hardware can provide useful feedback to the designers and manufacturers of trusted hardware. The holy grail would be to design and implement a formally secure processor. Understanding what cryptography-level formal abstraction to realize is a necessary first step towards this longer-term goal — but to realize this goal would obviously require additional, complementary techniques and machinery, e.g., those developed in the formal methods community [39, 65, 66, 81], that can potentially allow us to ensure that the actual secure processor implementation meets the specification.

1.3 Summary of Our Contributions

To the best of our knowledge, we are the first to investigate cryptographically sound and composable formal abstractions for realistic, attested execution secure processors. Our findings demonstrate both the "expected" and the (perhaps) "surprising".

The expected and the surprising. On one hand, we show that attested execution processors are indeed extremely powerful as one might have expected, and allow us to realize primitives that otherwise would have been impossible even when assuming stateless hardware tokens or virtual blackbox secure cryptographic obfuscation.

On the other hand, our investigation unveils subtle technical details that could have been easily overlooked absent an effort at formal modeling, and we draw several conclusions that might have come off as surprising initially (but of course, natural in hindsight). For example,

- We show that universally composable two-party computation is impossible if a single party does not have such a secure processor (and the other party does);

This was initially surprising to us, since we commonly think of an attested execution processor as offering an "omnipotent" trusted third party that can compute general-purpose, user-defined programs. When such a trusted third party exists, it would appear that any function can be evaluated securely and non-interactively, hiding both the program and data. One way to interpret our findings is that such intuitions are technically imprecise and dangerous to presume — while attested execution processors indeed come close to offering such a "trusted third party" ideal abstraction, there are aspects that are "imperfect" about this ideal abstraction that should not be overlooked, and a rigorous approach is necessary towards formally correct usage of trusted hardware.

Additional results for multi-party computation. We additionally show the following results:

- Universally composable two-party computation is indeed possible when both parties are equipped with an attested execution processor. We give an explicit construction and show that there are several interesting technicalities in its design and proof (which we shall comment on soon). Dealing with these technicalities also demonstrates how a *provably* secure protocol candidate would differ in important details from the most natural protocol candidates [49, 64, 70] practitioners would have adopted (which are not known to have provable composable security). This confirms the importance of formal modeling and provable security.

- Despite the infeasibility of multi-party computation when even a single party does not have a secure processor, in practice it would nonetheless be desirable to build multi-party applications where multiple possibly legacy clients outsource data and computation to a single cloud server equipped with a secure processor.

  We show how to introduce minimal global setup assumptions — more specifically, by adopting a global augmented common reference string [22] (henceforth denoted $\mathcal{G}_{\text{acrs}}$) — to circumvent this impossibility. Although the theoretical feasibility of general UC-secure MPC is known with $\mathcal{G}_{\text{acrs}}$ even without secure processors [22], existing constructions involve cryptographic computation that is (at least) linear in the runtime of the program to be securely evaluated. By contrast, we are specifically interested in *practical* constructions that involve only $O(1)$ amount of cryptographic computations, and instead perform all program-dependent computations inside the secure processor (and not cryptographically).

Techniques. Several interesting technicalities arise in our constructions. First, composition-style proofs typically require that a simulator intercepts and modifies communication to and from the adversary (and the environment), such that the adversary cannot distinguish whether it is talking to the simulator or the real-world honest parties and secure processors. Since the simulator does not know honest parties' inputs (beyond what is leaked by the computation output), due to the indistinguishability, one can conclude that the adversary cannot have knowledge of honest parties inputs either.

- *Equivocation.* Our simulator's ability to perform such simulation is hampered by the fact that the secure processors sign attestations for messages coming out — since the simulator does not

possess the secret signing key, it cannot modify these messages and must directly forward them to the adversary. To get around this issue would require new techniques for performing *equivocation*, a technicality that arises in standard protocol composition proofs. To achieve equivocation, we propose new techniques that place special backdoors inside the enclave program. Such backdoors must be carefully crafted such that they give the simulator more power without giving the real-world adversary additional power. In this way, we get the best of both worlds: 1) honest parties' security will not be harmed in the real-world execution; and 2) the simulator in the proof can "program" the enclave application to sign any output of its choice, provided that it can demonstrate the correct trapdoors. This technique is repeatedly used in different forms in almost all of our protocols.

- *Extraction.* Extraction is another technical issue that commonly arises in protocol composition proofs. The most interesting manifestation of this technical issue is in our protocol that realizes multi-party computation in the presence of a global common reference string ($\mathcal{G}_{\text{acrs}}$) and a single secure processor (see Section 6). Here again, we leverage the idea of planting special backdoors in the enclave program to allow for such extraction. Specifically, when provided with the correct identity key of a party, the enclave program will leak the party's inputs to the caller. Honest parties' security cannot be harmed by this backdoor, since no one ever learns honest parties' identity keys in the real world, not even the honest parties themselves. In the simulation, however, the simulator learns the corrupt parties' identity keys, and therefore it can extract corrupt parties' inputs.

Trusted clocks and fairness. Finally, we formally demonstrate how differences in abstraction can lead to differences in expressive power. In particular, many secure processors provide a trusted clock, and we explore the expressive power of such a trusted clock in the context of *fair* 2-party computation. It is well-known that in the standard setting fairness is impossible in 2-party computation for general functionalities [32]. However, several recent works have shown that the impossibility for general functionalities does not imply impossibility for every functionality — interestingly, there exist a broad class of functionalities that can be fairly computed in the plain setting [9, 46, 47]. We demonstrate several interesting findings in the context of attested execution processors:

- First, even a single attested execution processor already allows us to compute more functionalities fairly than in the plain setting. Specifically, we show that fair two-party coin flipping, which is impossible in the plain setting, is possible if only one party is equipped with an attested execution processor.
- Unfortunately, we show that a single attested execution processor is insufficient for fairly computing general 2-party functionalities;
- On the bright side, we prove that if both parties are equipped with an attested execution processor, it is indeed possible to securely compute any function fairly.

Variant models and additional results. Besides the trusted clock, we also explore variations in abstraction and their implications — for example, we compare non-anonymous attestation and anonymous attestation since various processors seem to make different choices regarding this.

We also explore an interesting model called "transparent enclaves" [79], where secret data inside the enclave can leak to the adversary due to possible side-channel attacks on known secure

processors, and we show how to realize interesting tasks such as UC-secure commitments and zero-knowledge proofs in this weaker model — here again our protocols must deal with interesting technicalities related to extraction and equivocation.

1.4 Non-Goals and Frequently Asked Questions

Trusted hardware has been investigated by multiple communities from different angles, ranging from how to architect secure processors [7, 28, 34, 39, 40, 57, 58, 60, 75, 81], how to apply them in applications [13,14,29,35,59,64,67,69–71], side-channels and other attacks [44,55,56,76,80,83] and protection against such attacks [40, 58, 81, 83]. Despite the extensive literature, cryptographically sound formal abstractions appear to be an important missing piece, and this work aims to make an initial step forward towards this direction. In light of the extensive literature, however, several natural but frequently asked questions arise regarding the precise scope of this paper, and we address such questions below.

First, although we base our modeling upon what realistic secure processors aim to provide, it is not our intention to claim that any existing secure processors provably realize our abstraction. We stress that to make any claim of this nature (that a secure processor correctly realizes any formal specification) is an area of active research in the formal methods and programming language communities [39, 65, 66, 81], and thus still a challenging open question — let alone the fact that some commercial secure processor designs are closed-source.

Second, a frequently asked question is what adversarial models our formal abstraction defends against. The answer to such a question is processor-specific, and thus outside the scope of our paper — we leave it to the secure processor itself to articulate the precise adversarial capabilities it protects against. The formal models and security theorems in this paper hold assuming that the adversary is indeed confined to the capabilities assumed by the specific secure processor. As mentioned earlier, some processors defend only against software adversaries [34]; others additionally defend against physical attackers [40–42, 58]; others defend against a restricted class of software and/or physical attackers that do not exploit certain side channels [2, 7, 57, 60, 75]. We refer the reader to a comprehensive systematization of knowledge paper by Shi et al. [72] for a taxonomy and comparison of various secure processors.

Finally, it is also not our goal to propose new techniques that defend against side-channel attacks, or suggest how to better architect secure processors — these questions are being explored in an orthogonal but complementary line of research [34, 39–42, 58, 81, 83].

2 Technical Roadmap

2.1 Formal Modeling

Modeling choices. To enable cryptographically sound reasoning, we adopt the universal composition (UC) paradigm in our modeling [21, 22, 26]. At a high level, the UC framework allows us to abstract complex cryptographic systems as simple ideal functionalities, such that protocol composition can be modularized. The UC framework also provides what is commonly referred to as "concurrent composition" and "environmental friendliness": in essence, a protocol π proven secure in the UC framework can run in any environment such that 1) any other programs or protocols executing possibly simultaneously will not affect the security of the protocol π, and 2) protocol π

will not inject undesirable side effects (besides those declared explicitly in the ideal abstraction) that would affect other programs and protocols in the system.

More intuitively, if a system involving cryptography UC-realizes some ideal functionality, henceforth, a programmer can simply program the system pretending that he is making remote procedural calls to a trusted third party without having to understand the concrete cryptography implementations. We refer the reader to Appendix A for a more detailed overview of the UC framework. Before we proceed, we stress the importance of cryptographically sound reasoning: by contrast, earlier works in the formal methods community would make assumptions that cryptographic primitives such as encryption and signatures realize the "most natural" ideal box without formal justification — and such approaches have been shown to be flawed when the ideal box is actually instantiated with cryptography [3–5, 10, 16, 24, 51, 53, 62, 63].

Roadmap for formal modeling. We first describe an ideal functionality $\mathcal{G}_{\text{att}}$ that captures the core abstraction that a broad class of attested execution processors intend to provide. We are well aware that various attested execution processors make different design choices — most of them are implementation-level details that do not reflect at the abstraction level, but a few choices do matter at the abstraction level — such as whether the secure processor provides a trusted clock and whether it implements anonymous or non-anonymous attestation.

In light of such differences, we first describe a basic, anonymous attestation abstraction called $\mathcal{G}_{\text{att}}$ that lies at the core of off-the-shelf secure processors such as Intel SGX [7, 60]. We explore the expressive power of this basic abstraction in the context of stateful obfuscation and multi-party computation. Later in the paper, we explore variants of the abstraction such as non-anonymous attestation and trusted clocks. Therefore, in summary our results aim to be broadly applicable to a wide class of secure processor designs.

The $\mathcal{G}_{\text{att}}$ abstraction. We first describe a basic $\mathcal{G}_{\text{att}}$ abstraction capturing the essence of SGX-like secure processors that provide anonymous attestation (see Figure 1). Here we briefly review the $\mathcal{G}_{\text{att}}$ abstraction and explain the technicalities that arise in the formal modeling, but defer more detailed discussions to Section 3.

1. **Registry.** First, $\mathcal{G}_{\text{att}}$ is parametrized with a registry `reg` that is meant to capture all the platforms that are equipped with an attested execution processor. For simplicity, we consider a static registry `reg` in this paper.

2. **Stateful enclave operations.** A platform $\mathcal{P}$ that is in the registry `reg` may invoke enclave operations, including

   - `install`: installing a new enclave with a program prog, henceforth referred to as the enclave program. Upon installation, $\mathcal{G}_{\text{att}}$ simply generates a fresh enclave identifier *eid* and returns the *eid*. This enclave identifier may now be used to uniquely identify the enclave instance.
   - `resume`: resuming the execution of an existing enclave with inputs inp. Upon a resume call, $\mathcal{G}_{\text{att}}$ executes the prog over the inputs inp, and obtains an output outp. $\mathcal{G}_{\text{att}}$ would then sign the prog together with outp as well as additional metadata, and return both outp and the resulting attestation.

$\mathcal{G}_{\text{att}}[\Sigma, \mathsf{reg}]$

// initialization:
On initialize: $(\mathsf{mpk}, \mathsf{msk}) := \Sigma.\mathsf{KeyGen}(1^\lambda)$, $T = \emptyset$

// public query interface:
On receive* `getpk`() from some $\mathcal{P}$: send mpk to $\mathcal{P}$

Enclave operations

// local interface — install an enclave:
On receive* `install`(idx, prog) from some $\mathcal{P} \in \mathsf{reg}$:
if $\mathcal{P}$ is honest, assert $idx = sid$
generate nonce $eid \in \{0,1\}^\lambda$, store $T[eid, \mathcal{P}] := (idx, \mathsf{prog}, \vec{0})$, send eid to $\mathcal{P}$

// local interface — resume an enclave:
On receive* `resume`(eid, inp) from some $\mathcal{P} \in \mathsf{reg}$:
let $(idx, \mathsf{prog}, \mathsf{mem}) := T[eid, \mathcal{P}]$, abort if not found
let $(\mathsf{outp}, \mathsf{mem}) := \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[eid, \mathcal{P}] := (idx, \mathsf{prog}, \mathsf{mem})$
let $\sigma := \Sigma.\mathsf{Sig}_{\mathsf{msk}}(idx, eid, \mathsf{prog}, \mathsf{outp})$, and send (outp, σ) to $\mathcal{P}$

Figure 1: A global functionality modeling an SGX-like secure processor. Blue (and starred*) activation points denote *reentrant* activation points. Green activation points are executed at most once. The enclave program prog may be probabilistic and this is important for privacy-preserving applications. Enclave program outputs are included in an anonymous attestation σ. For honest parties, the functionality verifies that installed enclaves are parametrized by the session id sid of the current protocol instance.

Each installed enclave can be resumed multiple times, and we stress that the enclave operations store state across multiple `resume` invocations. This stateful property will later turn out to be important for several of our applications.

3. **Anonymous attestation.** Secure processors such as SGX rely on group signatures and other anonymous credential techniques [18, 19] to offer "anonymous attestation". Roughly speaking, anonymous attestation allows a user to verify that the attestation is produced by some attested execution processor, without identifying which one. To capture such anonymous attestation, our $\mathcal{G}_{\text{att}}$ functionality has a manufacturer public key and secret key pair denoted $(\mathsf{mpk}, \mathsf{msk})$, and is parametrized by a signature scheme Σ. When an enclave `resume` operation is invoked, $\mathcal{G}_{\text{att}}$ signs any output to be attested with msk using the signature scheme Σ. Roughly speaking, if a group signature scheme is adopted as in SGX, one can think of Σ as the group signature scheme parametrized with the "canonical" signing key. $\mathcal{G}_{\text{att}}$ provides the manufacturer public key mpk to any party upon query — this models the fact that there exists a secure key distribution channel to distribute mpk. In this way, any party can verify an anonymous attestation signed by $\mathcal{G}_{\text{att}}$.

Globally shared functionality. Our $\mathcal{G}_{\text{att}}$ functionality essentially captures all attested execution processors in the world. Further, we stress that $\mathcal{G}_{\text{att}}$ is globally shared by all users, all applications, and all protocols. In particular, rather than generating a different $(\mathsf{mpk}, \mathsf{msk})$ pair for each different protocol instance, the same $(\mathsf{mpk}, \mathsf{msk})$ pair is globally shared.

More technically, we capture such sharing across protocols using the Universal Composition with Global Setup (GUC) paradigm [22]. As we show later, such global sharing of cryptographic keys becomes a source of "imperfectness" — in particular, due to the sharing of $(\mathsf{mpk}, \mathsf{msk})$, attestations signed by msk from one protocol instance (i.e., or application) may now carry meaning in a completely unrelated protocol instance, thus introducing potentially undesirable side effects that breaks composition.

Additional discussions and clarifications. We defer more detailed discussions of our modeling choices, and importantly, clarifications on how the environment $\mathcal{Z}$ interacts with $\mathcal{G}_{\text{att}}$ to Section 3.

Throughout this paper, we assume that parties interact with each other over *secure* channels. It is possible to realize (UC-secure) secure channels from authenticated channels through key exchange. Whenever applicable, our results are stated for the case of *static* corruption.

2.2 Power of Attested Execution: Stateful Obfuscation

We show that the attested execution abstraction is indeed extremely powerful as one would have expected. In particular, we show that attested execution processors allow us to realize a new abstraction which we call "stateful obfuscation".

Theorem 1 (Informal)**.** *Assume that secure key exchange protocols exist. There is a $\mathcal{G}_{att}$-hybrid protocol that realizes non-interactive stateful obfuscation, which is not possible in plain settings, even when assuming stateless hardware tokens or virtual-blackbox secure cryptographic obfuscation.*

Stateful obfuscation allows an (honest) client to obfuscate a program and send it to a server, such that the server can evaluate the obfuscated program on multiple inputs, while the obfuscated program keeps (secret) internal state across multiple invocations. We consider a simulation secure notion of stateful obfuscation, where the server should learn only as much information as if it were interacting with a stateful oracle (implementing the obfuscated program) that answers the server's queries. For example, stateful obfuscation can be a useful primitive in the following application scenario: imagine that a client (e.g., a hospital) outsources a sensitive database (corresponding to the program we wish to obfuscate) to a cloud server equipped with trusted hardware. Now, an analyst may send statistical queries to the server and obtain *differentially private* answers. Since each query consumes some privacy budget, we wish to guarantee that after the budget is depleted, any additional query to the database would return $\perp$. We formally show how to realize stateful obfuscation from attested execution processors. Further, as mentioned, we prove that stateful obfuscation is not possible in the plain setting, even when assuming the existence of stateless hardware tokens or assuming virtual-blackbox secure obfuscation.

2.3 Impossibility of Composable 2-Party Computation with a Single Secure Processor

One natural question to ask is whether we can realize universally composable (i.e., UC-secure) multi-party computation, which is known to be impossible in the plain setting without any setup

assumptions — but feasible in the presence of a common reference string [21, 23], i.e., a public random string that is generated in a trustworthy manner freshly and independently for each protocol instance. On the surface, $\mathcal{G}_{\text{att}}$ seems to provide a much more powerful functionality than a common reference string, and thus it is natural to expect that it will enable UC-secure multi-party computation. However, upon closer examination, we find that perhaps somewhat surprisingly, such intuition is subtly incorrect, as captured in the following informal theorem.

Theorem 2 (Informal)**.** *If at least one party is not equipped with an attested execution processor, it is impossible to realize UC-secure multi-party computation absent additional setup assumptions (even when all others are equipped with an attested execution processor).*

Here the subtle technicalities arise exactly from the fact that $\mathcal{G}_{\text{att}}$ is a global functionality shared across all users, applications, and protocol instances. This creates a *non-deniability* issue that is well-known to the cryptography community. Since the manufacturer signature key $(\mathsf{mpk}, \mathsf{msk})$ is globally shared, attestations produced in one protocol instance can carry side effects into another. Thus, most natural protocol candidates that send attestations to other parties will allow an adversary to implicate an honest party of having participated in a protocol, by demonstrating the attestation to a third party. Further, such non-deniability exists even when the secure processor signs *anonymous* attestations: since if not all parties have a secure processor, the adversary can at least prove that *some* honest party that is in $\mathcal{G}_{\text{att}}$'s registry has participated in the protocol, even if he cannot prove which one. Intuitively, the non-deniability goes away if all parties are equipped with a secure processor — note that this necessarily means that the adversary himself must have a secure processor too. Since the attestation is anonymous, the adversary will fail to prove whether the attestation is produced by an honest party or he simply asked his own local processor to sign the attestation. This essentially allows the honest party to deny participation in a protocol.

Impossibility of extraction. We formalize the above intuition, and show that not only natural protocol candidates that send attestations around suffer from non-deniability, in fact, it is impossible to realize UC-secure multi-party computation if not all parties have secure processors. The impossibility is analogous to the impossibility of UC-secure commitments in the plain setting absent a common reference string [23]. Consider when the real-world committer $\mathcal{C}$ is corrupt and the receiver is honest. In this case, during the simulation proof, when the real-world $\mathcal{C}$ outputs a commitment, the ideal-world simulator Sim must capture the corresponding transcripts and extract the value v committed, and send v to the commitment ideal functionality $\mathcal{F}_{\text{com}}$. However, if the ideal-world simulator Sim can perform such extraction, the real-world receiver must be able too (since Sim does not have extra power than the real-world receiver) — and this violates the requirement that the commitment must be hiding. As Canetti and Fischlin show [23], a common reference string allows us to circumvent this impossibility by giving the simulator more power. Since a common reference string (CRS) is a *local* functionality, during the simulation, the simulator can program the CRS and embed a trapdoor — this trapdoor will allow the simulator to perform extraction. Since the real-world receiver does not possess such a trapdoor, the protocol still retains confidentiality against a real-world receiver.

Indeed, if our $\mathcal{G}_{\text{att}}$ functionality were also local, our simulator Sim could have programmed $\mathcal{G}_{\text{att}}$ in a similar manner and extraction would have been easy. In practice, however, a local $\mathcal{G}_{\text{att}}$ function would mean that a fresh key manufacturer pair $(\mathsf{mpk}, \mathsf{msk})$ must be generated for each protocol instance (i.e., even for multiple applications of the same user). Thus, a local $\mathcal{G}_{\text{att}}$ clearly fails to

$\mathsf{prog}_{\mathrm{2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, b]$

On input ("keyex"): $y \xleftarrow{\$} \mathbb{Z}_p$, return g^y

On input ("send", g^x, inp_b):
  assert that "keyex" has been called
  $\mathsf{sk} := (g^x)^y$, $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(\mathsf{inp}_b)$, return ct

On input ("compute", ct, v):
  assert that "send" has been called and ct not seen
  $\mathsf{inp}_{1-b} := \mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct})$, assert that decryption succeeds
  if $v \neq \perp$, return v; else return $\mathsf{outp} := f(\mathsf{inp}_0, \mathsf{inp}_1)$

$\mathbf{Prot}_{\mathrm{2pc}}[sid, f, \mathcal{P}_0, \mathcal{P}_1, b]$

On input inp_b from $\mathcal{Z}$:
  $eid := \mathcal{G}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathrm{2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, b])$
  henceforth denote $\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\cdot) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(eid, \cdot)$
  $(g^y, \sigma) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"keyex"})$
  send (eid, g^y, σ) to $\mathcal{P}_{1-b}$, await (eid', g^x, σ')
  assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((sid, eid', \mathsf{prog}_{\mathrm{2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, 1-b], g^x), \sigma')$
  $(\mathsf{ct}, \_) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"send"}, g^x, \mathsf{inp}_b)$, send ct to $\mathcal{P}_{1-b}$, await ct′
  $(\mathsf{outp}, \_) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"compute"}, \mathsf{ct}', \perp)$, output outp

Figure 2: Composable 2-party computation: both parties have secure processors. AE denotes authenticated encryption. All ITIs' activation points are *non-reentrant*. When an activation point is invoked for more than once, the ITI simply outputs $\perp$. Although not explicitly noted, if $\mathcal{G}_{\mathrm{att}}$ ever outputs $\perp$ upon a query, the protocol aborts outputting $\perp$. The group parameters (g, p) are hardcoded into $\mathsf{prog}_{\mathrm{2pc}}$.

capture the reusability of real-world secure processors, and this justifies why we model attested execution processors as a globally shared functionality.

Unfortunately, when $\mathcal{G}_{\mathrm{att}}$ is global, it turns out that the same impossibility of extraction from the plain setting would carry over when the committer $\mathcal{C}$ is corrupt and only the receiver has a secure processor. In this case, the simulator Sim would also have to extract the input committed from transcripts emitted from $\mathcal{C}$. However, if the simulator Sim can perform such extraction, so can the real-world receiver — note that in this case the real-world receiver is actually more powerful than Sim, since the real-world receiver, who is in the registry, is capable of meaningfully invoking $\mathcal{G}_{\mathrm{att}}$, while the simulator Sim cannot!

It is easy to observe that this impossibility result no longer holds when the corrupt committer has a secure processor — in this case, the protocol can require that the committer $\mathcal{C}$ send its input to $\mathcal{G}_{\mathrm{att}}$. Since the simulator captures all transcripts going in and coming out of $\mathcal{C}$, it can extract the input trivially. Indeed, we show that not only commitment, but also general 2-party computation is possible when both parties have a secure processor.

2.4 Composable 2-Party Computation When Both Have Secure Processors

Theorem 3 (Informal)**.** *Assume that secure key exchange protocols exist. Then there exists an $\mathcal{G}_{att}$-hybrid protocol that UC-realizes $\mathcal{F}_{2pc}$. Further, in this protocol, all program-dependent evaluation is performed inside the enclave and not cryptographically.*

We give an explicit protocol in Figure 2 (for concreteness, we use Diffie-Hellman key exchanges in our protocols, although the same approach extends to any secure key-exchange). The protocol is efficient in the sense that it performs only $O(1)$ (program-independent) cryptographic computations; and all program-dependent computation is performed inside the enclave. We now explain the protocol briefly.

- First, the two parties' secure processors perform a key exchange and establish a secret key sk for an authenticated encryption scheme.
- Then, each party's enclave encrypts the party's input with sk. The party then sends the resulting authenticated ciphertext ct to the other.
- Now each enclave decrypts ct and perform evaluation, and each party can query its local enclave to obtain the output.
- Most of the protocol is quite natural, but one technique is necessary for equivocation. Specifically, the enclave program's "compute" entry point has a backdoor denoted v. If $v = \perp$, $\mathcal{G}_{\text{att}}$ will sign the true evaluation result and return the attested result. On the other hand, if $v \neq \perp$, the enclave will simply sign and output v itself. In the real-world execution, an honest party will always supply $v = \perp$ as input to the enclave program's "compute" entry point. However, as we explain later, the simulator will leverage this backdoor v to perform equivocation and program the output.

We now explain some interesting technicalities that arise in the proof for the above protocol.

- *Extraction.* First, extraction is made possible since each party sends their input directly to its local enclave. If a party is corrupt, this interaction will be captured by the simulator who can then extract the corrupt party's input;
- *Equivocate.* We now explain how the backdoor v in the enclave program allows for equivocation in the proof. Recall that initially, the simulator does not know the honest party's input. To simulate the honest party's message for the adversary (which contains an attestation from the enclave), the simulator must send a dummy input to $\mathcal{G}_{\text{att}}$ on behalf of the honest party to obtain the attestation. When the simulator manages to extract the corrupt party's input, it will send the input to the ideal functionality $\mathcal{F}_{\text{2pc}}$ and obtain the outcome of the computation denoted outp^*. Now when the corrupt party queries its local enclave for the output, the simulator must get $\mathcal{G}_{\text{att}}$ to sign the correct outp^* (commonly referred to as *equivocation*). To achieve this, the simulator will make use of the aforementioned backdoor v: instead of sending $(\mathsf{ct}, \perp)$ to $\mathcal{G}_{\text{att}}$ as in the real-world protocol, the simulator sends $(\mathsf{ct}, \mathsf{outp}^*)$ to $\mathcal{G}_{\text{att}}$, such that $\mathcal{G}_{\text{att}}$ will sign outp^*.
- *A note on anonymous attestation.* It is interesting to note how our protocol relies on the attestation being *anonymous* for security. Specifically, in the proof, the simulator needs to simulate

the honest party's messages for the adversary $\mathcal{A}$. To do so, the simulator will simulate the honest party's enclave on its own (i.e., the adversary's) secure processor — and such simulation is possible because the attestations returned by $\mathcal{G}_{\text{att}}$ are anonymous. Had the attestation not been anonymous (e.g., binding to the party's identifier), the simulator would not be able to simulate the honest party's enclave (see Section 8.4 for more discussions).

2.5 Circumventing the Impossibility with Minimal Global Setup

In practice, it would obviously be desirable if we could allow composable multi-party computation in the presence of a single attested execution processor. As a desirable use case, imagine multiple clients (e.g., hospitals), each with sensitive data (e.g., medical records), that wish to perform some computation (e.g., data mining for clinical research) over their joint data. Moreover, they wish to outsource the data and computation to an untrusted third-party cloud provider. Specifically, the clients may not have secure processors, but as long as the cloud server does, we wish to allow outsourced secure multi-party computation.

We now demonstrate how to introduce a minimal global setup assumption to circumvent this impossibility. Specifically, we will leverage a global Augmented Common Reference String (ACRS) [22], henceforth denoted $\mathcal{G}_{\text{acrs}}$. Although the feasibility of UC-secure multi-party computation is known with $\mathcal{G}_{\text{acrs}}$ even absent secure processors [22], existing protocols involve cryptographic computations that are (at least) linear in the runtime of the program. Our goal is to demonstrate a *practical* protocol that performs any program-dependent computation inside the secure enclave, and performs only $O(1)$ cryptographic computation.

Theorem 4 (Informal)**.** *Assume that secure key exchange protocols exist. Then, there exists a* $(\mathcal{G}_{\text{acrs}}, \mathcal{G}_{att})$*-hybrid protocol that UC-realizes* $\mathcal{F}_{\text{mpc}}$ *and makes use of only a single secure processor. Further, this protocol performs all program-dependent computations inside the secure processor's enclave (and not cryptographically).*

Minimal global setup $\mathcal{G}_{\text{acrs}}$. To understand this result, we first explain the minimal global setup $\mathcal{G}_{\text{acrs}}$. First, $\mathcal{G}_{\text{acrs}}$ provides a global common reference string. Second, $\mathcal{G}_{\text{acrs}}$ also allows each (corrupt) party $\mathcal{P}$ to query an *identity key* for itself. This identity key is computed by signing the party's identifier $\mathcal{P}$ using a global master secret key. Note that such a global setup is *minimal* since *honest parties should never have to query for their identity keys.* The identity key is simply a backdoor provided to corrupt parties. Although at first sight, it might seem counter-intuitive to provide a backdoor to the adversary, note that this backdoor is also provided to our simulator — and this increases the power of the simulator allowing us to circumvent the aforementioned impossibility of extraction, and design protocols where honest parties can deny participation.

MPC with a single secure processor and $\mathcal{G}_{\text{acrs}}$. We consider a setting with a server that is equipped with a secure processor, and multiple clients that do not have a secure processor.

Let us first focus on the (more interesting) case when the server and a subset of the clients are corrupt. The key question is how to get around the impossibility of extraction with the help of $\mathcal{G}_{\text{acrs}}$ — more specifically, how does the simulator extract the corrupt clients' inputs? Our idea is the following — for the readers' convenience, we skip ahead and present the detailed protocol in Figure 3 as we explain the technicalities, but we will revisit it and present formal notations and proofs in Section 6.

$\mathsf{prog}_{\mathrm{mpc}}[f, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n]$

On input ("init"): for $i \in [n]$: $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$; return $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$

On input ("input", $\{\mathsf{ct}_i\}_{i\in[n]}$):
for $i \in [n]$: $(\mathsf{inp}_i, k_i) := \mathsf{PKE.Dec}_{\mathsf{sk}_i}(\mathsf{ct}_i)$; return $\Omega := \{\mathsf{ct}_i\}_{i\in[n]}$

On input ("extract", $\{\mathsf{idk}_i\}_{i\in[n]}$):
for $i \in [n]$: if $\mathsf{check}(\mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{P}_i, \mathsf{idk}) = 1$, $v_i := \mathsf{sk}_i$, else $v_i := \perp$; return $\{v_i\}_{i\in[n]}$

On input ("program", $\{\mathsf{idk}_i, u_i\}_{i\in[n]}$):
for $i \in [n]$: if $\mathsf{check}(\mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{P}_i, \mathsf{idk}) = 1$, $\mathsf{outp}_i := u_i$

On input ("proceed", $\{\mathsf{ct}'_i\}_{i\in[n]}$):
for $i \in [n]$: assert $\mathsf{AE.Dec}_{k_i}(\mathsf{ct}'_i) =$ "ok"
$\mathsf{outp}^* := f(\mathsf{inp}_1, \ldots, \mathsf{inp}_n)$, return "done"

On input* ("output", $\mathcal{P}_i$):
assert outp^* has been stored
if outp_i has been stored, $\mathsf{ct} := \mathsf{Enc}_{k_i}(\mathsf{outp}_i)$, else $\mathsf{ct} := \mathsf{Enc}_{k_i}(\mathsf{outp}^*)$
return ct

$\mathbf{Prot}_{\mathrm{mpc}}[sid, f, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n]$

Server $\mathcal{S}$:

let $eid := \mathcal{G}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathrm{mpc}}[f, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n])$
henceforth let $\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\cdot) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(eid, \cdot)$
let $(\{\mathsf{pk}_i\}_{i\in[n]}, \sigma) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"init"})$, send $(eid, \psi(\mathcal{P}_i, \{\mathsf{pk}_i\}_{i\in[n]}, \sigma))$ to each $\mathcal{P}_i$

for each $\mathcal{P}_i$: await ("input", ct_i) from $\mathcal{P}_i$
$(\Omega, \sigma) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"input"}, \{\mathsf{ct}_i\}_{i\in[n]})$, send $\psi(\mathcal{P}_i, \Omega, \sigma)$ to each $\mathcal{P}_i$

for each $\mathcal{P}_i$: await ("proceed", ct'_i) from $\mathcal{P}_i$
$\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"proceed"}, \{\mathsf{ct}'_i\}_{i\in[n]})$
for each $\mathcal{P}_i$: $(\mathsf{ct}_i, \sigma_i) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"output"}, \mathcal{P}_i)$, send ct_i to $\mathcal{P}_i$

Remote Party $\mathcal{P}_i$: On input inp from $\mathcal{Z}$:

await (eid, ψ) from $\mathcal{S}$
// Henceforth for $\widetilde{\psi} := (\mathsf{msg}, C, \pi)$, let $\mathsf{Ver}(\widetilde{\psi}) := \mathsf{Ver}(\mathsf{crs}, (sid, eid, C, \mathsf{mpk}, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{P}_i, \mathsf{msg}), \pi)$
assert $\mathsf{Ver}(\psi)$, parse $\psi := (\{\mathsf{pk}_i\}_{i\in[n]}, \_, \_)$
$k \leftarrow \{0,1\}^\lambda$, $\mathsf{ct} = \mathsf{PKE.Enc}_{\mathsf{pk}}(\mathsf{inp}, k)$ where $\mathsf{pk} := \mathsf{pk}_i$
send ("input", ct) to $\mathcal{S}$, await ψ from $\mathcal{S}$, assert $\mathsf{Ver}(\psi)$, parse $\psi := (\Omega, \_, \_)$
assert $\Omega[i] = \mathsf{ct}$, send eid to all parties, wait for all parties to ack the same eid
let $\mathsf{ct}' := \mathsf{AE.Enc}_k(\text{"ok"})$, send ("proceed", ct') to $\mathcal{S}$, await ct, assert ct not seen
$\mathsf{outp} := \mathsf{Dec}_k(\mathsf{ct})$, assert ct decryption successful, return outp

Figure 3: Composable multi-party computation with a single secure processor. $\psi(\mathcal{P}, \mathsf{msg}, \sigma)$ outputs a tuple (msg, C, π), where π is a witness-indistinguishable proof that the ciphertext C either encrypts a valid attestation σ on msg, or encrypts $\mathcal{P}$'s identity key. PKE and AE denote public-key encryption and authenticated encryption respectively. The notation send denotes messages sent over a secure channel.

- First, we parametrize the enclave program with the global common reference string $\mathcal{G}_{\text{acrs}}.\mathsf{mpk}$.
- Second, we add a backdoor in the enclave program, such that the enclave program will return the secret key for $\mathcal{P}_i$'s secure channel with the enclave, if the caller provides the correct identity key for $\mathcal{P}_i$. In this way, the simulator can be a man-in-the-middle for all corrupt parties' secure channels with the enclave, and extract their inputs. We note that honest parties' security will not be harmed by this backdoor, since honest parties will never even query $\mathcal{G}_{\text{acrs}}$ for their identity keys, and thus their identity keys should never leak. However, in the simulation, the simulator will query $\mathcal{G}_{\text{acrs}}$ for all corrupt parties' identity keys, which will allow the simulator to extract corrupt parties' inputs by querying this backdoor in the enclave program.
- Third, we introduce yet another backdoor in the enclave program that allows the caller to program any party's output, provided that the caller can demonstrate that party's identity key. Again, in the real world, this backdoor should not harm honest parties' security because honest parties' identity keys never get leaked. Now in the simulation, the simulator will query $\mathcal{G}_{\text{acrs}}$ for all corrupt parties' identity keys which will give the simulator the power to query the corrupt parties' outputs. Such "programmability" is necessary, because when the simulator obtains the outcome outp from $\mathcal{F}_{\text{mpc}}$, it must somehow obtain the enclave's attestation on outp — however, since the simulator does not know honest parties' inputs, he cannot have provided honest parties' inputs to the enclave. Therefore, there must be a special execution path such that the simulator can obtain a signature on outp from the enclave.

Now, let us turn our attention to the case when the server is honest, but a subset of the clients are corrupt. In this case, our concern is how to achieve *deniability* for the server — specifically, an honest server should be able to deny participation in a protocol. If the honest server sends an attestation in the clear to the (possibly corrupt) clients, we cannot hope to obtain such deniability, because a corrupt client can then prove to others that *some* honest party in $\mathcal{G}_{\text{att}}$'s registry must have participated, although it might not be able to prove which one since the attestation is anonymous. To achieve deniability, our idea is the following:

- Instead of directly sending an attestation on a message msg, the server will produce a witness indistinguishable proof that either he knows an attestation on msg, or he knows the recipient's identity key. Note that in the real world protocol, the server always provide the attestation as the witness when producing the witness indistinguishable proof.
- However, in the simulation when the server is honest but a subset of the clients are corrupt, the simulator is unable to query any enclave since none of the corrupt clients have a secure processor. However, the simulator can query $\mathcal{G}_{\text{acrs}}$ and obtain all corrupt parties' identity keys. In this way, the simulator can use these identity keys as an alternative witness to construct the witness indistinguishable proofs — and the witness indistinguishability property ensures that the adversary (and the environment) cannot distinguish which witness was provided in constructing the proof.

Implementing $\mathcal{G}_{\text{acrs}}$. In practice, the $\mathcal{G}_{\text{acrs}}$ functionality can be implemented by having a trusted third party (which may be the trusted hardware manufacturer) that generates the reference string and hands out the appropriate secret keys [22].

It is instructive to consider why $\mathcal{G}_{\text{acrs}}$ cannot be implemented from $\mathcal{G}_{\text{att}}$ itself (indeed, this would contradict our result that it is impossible to obtain composable MPC in the presence of a single attested execution processor, with no further setup assumptions). Informally, the reason this does not work is that unless all parties have access to $\mathcal{G}_{\text{att}}$ (which is the case we consider), then if only the party that does not have access to $\mathcal{G}_{\text{att}}$ is corrupted, the view of the adversary cannot be simulated—in particular, the attested generation of the CRS cannot be simulated (since the adversary does not have access to $\mathcal{G}_{\text{att}}$) and as such serves as evidence that some honest party participated in an execution (i.e., we have a "deniability attack").

2.6 Fairness

It is well-known that fairness is in general impossible in secure two-party computation in the plain model (even under weaker security definitions that do not necessarily aim for concurrent composition). Intuitively, the party that obtains the output first can simply abort from the protocol thus preventing the other party from learning the outcome. Cleve [32] formalized this intuition and demonstrated an impossibility result for fair 2-party coin tossing, which in turns suggests the impossibility of fairness in general 2-party computation. Interestingly, a sequence of recent works show that although fairness is impossible in general, there are a class of non-trivial functions that can indeed be computed fairly [9, 46, 47].

Since real-world secure processors such as Intel's SGX offer a "trusted clock" abstraction, we explore whether and how such trusted clocks can help in attaining fairness. It is not hard to see that Cleve's lower bound still applies, and fairness is still impossible when our attested execution processors do not have trusted clocks. We show how having trusted clocks in secure processors can help with fairness.

First, we show that fairness is indeed possible in general 2-party computation, when both parties have secure processors with trusted clocks. Specifically, we consider a clock-adjusted notion of fairness which we refer to as Δ-fairness. Intuitively, Δ-fairness stipulates that if the corrupt party receives output by some round r, then the honest party must receive output by round $\Delta(r)$, where Δ is a polynomial function.

Theorem 5 (Informal)**.** *Assume that secure key exchange protocols exist, and that both parties have an attested execution processor with trusted clocks, then there exists a protocol that UC-realizes $\mathcal{F}_{\text{2pc}}$ with Δ-fairness where $\Delta(r) = 2r$.*

In other words, if the corrupt party learns the outcome by round r, the honest party is guaranteed to learn the outcome by round $2r$. Our protocol is a *tit-for-tat* style protocol that involves the two parties' enclaves negotiating with each other as to when to release the output to its owner. At a high level, the protocol works as follows:

- First, each party sends their respective input to its local secure processor.
- The two secure processors then perform a key exchange to establish a secret key k for an authenticated encryption scheme. Now the two enclave exchange the parties' inputs over a secure channel, at which point both enclaves can compute the output.
- However, at this point, the two enclaves still withhold the outcome from their respective owners, and the initial timeout value $\delta := 2^{\lambda}$ is set to exponentially large in λ. In other words, each enclave promises to release the outcome to its owner in round δ.

- At this moment, the *tit-for-tat* protocol starts. In each turn, each secure enclave sends an acknowledgment to the other over a secure channel. Upon receiving the other enclave's acknowledgment, the receiving enclave would now halve the δ value, i.e., set $\delta := \frac{\delta}{2}$. In other words, the enclave promises to release the outcome to its owner by half of the original timeout.

- If both parties are honest, then after λ turns, their respective enclaves disclose the outputs to each party.

- If one party is corrupt, then if he learns the outcome by round r, clearly the other party will learn the outcome by round $2r$.

To have provably security in the UC model, technicalities similar to our earlier 2-party computation protocol (the case when both parties have a secure processor) exist. More specifically, both parties have to send inputs to their local enclave to allow extraction in the simulation. Moreover, the enclave program needs to leave a second input (that is not used in the real-world protocol) such that the simulator can program the output for the corrupt party after learning the output from $\mathcal{F}_{\text{2pc}}$.

It is also worth noting that our protocol borrows ideas from gradual release-style protocols [17,38,43]. However, in comparison, known gradual release-style protocols rely on non-standard assumptions which are not necessary in our protocol when a clock-aware $\mathcal{G}_{\text{att}}$ is available.

We next consider whether a single secure processor enabled with trusted clock can help with fairness. We show two results: first, fairness is in impossible for generic functionalities when only one party has a clock-aware secure processor; and second, a single clock-aware secure processor allows us to fairly compute a broader class of functions than the plain setting.

Theorem 6 (Informal)**.** *Assume that one-way functions exist, then, fair 2-party computation is impossible for general functionalities when only one party has a clock-aware secure processor (even when assuming the existence of* $\mathcal{G}_{\text{acrs}}$*).*

First, to prove the general fairness impossibility in the presence of a single secure processor, we consider a specific contract signing functionality $\mathcal{F}_{\text{contract}}$ in which two parties, each with a secret signing key, exchange signatures over a canonical message, say $\vec{0}$ (see Section 7 for a formal definition). In the plain model, there exists a (folklore) fairness impossibility proof for this functionality — and it helps to understand this proof first before presenting ours. Imprecisely speaking, if one party, say $\mathcal{P}_0$, aborts prior to sending the last protocol message, and $\mathcal{P}_0$ is able to output a correct signature over the message, then $\mathcal{P}_1$ must be able to output the correct signature as well by fairness. As a result, we can remove protocol messages one by one, and show that if the previous protocol Π_i fairly realizes $\mathcal{F}_{\text{contract}}$, then Π_{i-1} (that is, the protocol Π_i with the last message removed) must fairly realize $\mathcal{F}_{\text{contract}}$ as well. Eventually, we will arrive at the empty protocol, and conclude that the empty protocol fairly realizes $\mathcal{F}_{\text{contract}}$ as well which clearly is impossible if the signature scheme is secure. Although the intuition is simple, it turns out that the formal proof is somewhat subtle — for example, clearly the proof should not work had this been some other functionality that is not contract signing, since we know that there exist certain functions that can be computed fairly in the plain model [9, 46, 47]. Therefore, we first formalize this folklore proof in Section 7.4 before presenting our own.

We now discuss how we can prove impossibility when only one party has a clock-aware secure processor. The overall structure of the proof is very similar to the aforementioned folklore proof

where protocol messages are removed one by one, however, as we do so, we need to carefully bound the time by which the corrupt (i.e., aborting) party learns output. Without loss of generality, let us assume that party $\mathcal{P}_0$ has a secure processor and party $\mathcal{P}_1$ does not. As we remove protocol messages one by one, in each alternate round, party $\mathcal{P}_1$ is the aborting party. Suppose party $\mathcal{P}_1$ aborts in round $r \leq g(\lambda)$ where $g(\lambda)$ is the runtime of the protocol if both parties are honest. Since $\mathcal{P}_1$ does not have a secure processor, if he can learn the result in polynomially many rounds by the honest protocol, then he must be able to learn the outcome in round r too — in particular, even if the honest protocol specifies that he waits for more rounds, he can just simulate the fast forwarding of his clock in a single round and complete the remainder of his execution. This means that as we remove protocol messages one by one, in every alternate turn, the aborting party is guaranteed to obtain output by round $g(\lambda)$ — and thus even if he aborts, the other party must receive output by round $\Delta(g(\lambda))$. Similar as before, we eventually arrive at an empty protocol which we conclude to also fairly compute $\mathcal{F}_{\text{contract}}$ (where the parties do not exchange protocol messages) which clearly is impossible if the signature scheme is secure.

We stress that the ability to reset the aborting party's runtime back to $g(\lambda)$ in every alternative round is important for the proof to work. In particular, if both parties have a clock-aware secure processor, the lower bound clearly should fail in light of our upper bound — and the reason that it fails is because the runtime of the aborting party would increase by a polynomial factor every time we remove a protocol message, and after polynomially many such removals the party's runtime would become exponential.

We also note that the above is simply the intuition, and formalizing the proof is somewhat subtle which we leave to Section 7.4.

Although fairness is impossible in general with only one clock-aware secure processor, we show that even one clock-aware secure processor can help with fairness too. Specifically, it broadens the set of functions that can be computed fairly in comparison with the plain setting.

Theorem 7 (Informal)**.** *Assume that secure key exchange protocols exist, then when only a single party has a clock-aware secure processor, there exist functions that can be computed with Δ-fairness in the $(\mathcal{G}_{att}, \mathcal{G}_{\text{acrs}})$-hybrid model, but cannot be computed fairly in the $\mathcal{G}_{\text{acrs}}$-hybrid model.*

Specifically, we show that 2-party fair coin toss, which is known to be impossible in the plain model, becomes possible when only one party has a clock-aware secure processor. Intuitively, the issue in the standard setting is that the party that obtains the output first can examine the outcome coin, and can abort if he does not like the result, say abort on 0. Although the other party can now toss another coin on his own — the first party aborting already suffices to bias the remaining party's output towards 1. We now propose a $(\mathcal{G}_{\text{att}}, \mathcal{G}_{\text{acrs}})$-hybrid protocol that realizes 2-party fair toss, assuming that $\mathcal{G}_{\text{att}}$ is clock aware and that only one party has a secure processor. The idea is the following. Let the server $\mathcal{S}$ and the client $\mathcal{C}$ be the two parties involved, and suppose that the server has a secure processor but the client does not. The server's enclave first performs key exchange and establishes a secure channel with the client. Now the server's enclave flips a random coin and sends it to the client over the secure channel in a specific round, say, round 3 (e.g., assuming that key exchange takes two rounds). At this moment, the server does not see the outcome of the coin yet. If the client does not receive this coin by the end of round 3, it will flip an independent coin on its own; otherwise it outputs the coin received. Finally, in round 4, the server will receive the outcome of the coin from its local enclave. Observe that server can decide to abort prior to sending the client the coin (over the secure channel), however, the server cannot base the decision upon the

value of the coin, since he does not get to see the coin until round 4. To formalize this intuition and specifically to prove the resulting protocol secure in the UC model, again we need to rely on the help of $\mathcal{G}_{\text{acrs}}$.

2.7 Additional Results

We provide some additional interesting variations in modeling and results.

The transparent enclave model. Many known secure processors are known to be vulnerable to certain side-channel attacks such as cache-timing or differential power analysis. Complete defense against such side channels remains an area of active research [39–42, 58, 81].

Recently, Tramèr et al. [79] ask the question, what kind of interesting applications can we realize assuming that such side-channels are unavoidable in secure processors? Tramèr et al. [79] then propose a new model which they call the *transparent enclave* model. The transparent enclave model is almost the same as our $\mathcal{G}_{\text{att}}$, except that the enclave program leaks all internal states to the adversary $\mathcal{A}$. Nonetheless, $\mathcal{G}_{\text{att}}$ still keeps its master signing key msk secret. In practice, this model requires us to only spend effort to protect the secure processor's attestation algorithm from side channels, and we consider the entire user-defined enclave program to be transparent to the adversary.

Tramèr et al. then show how to realize interesting security tasks such as cryptographic commitments and zero-knowledge proofs with only transparent enclaves. We note that Tramèr et al. adopt modeling techniques that inherit from an earlier manuscript version of the present paper. However, Tramèr et al. model $\mathcal{G}_{\text{att}}$ as a local functionality rather than a globally shared functionality — and this lets them circumvent several technical challenges that stem from the functionality being globally shared, and allow them to achieve universally composable protocols more trivially. As mentioned earlier, if $\mathcal{G}_{\text{att}}$ were local, in practice this would mean that a fresh $(\mathsf{mpk}, \mathsf{msk})$ pair is generated for every protocol instance — even for different applications of the same user. This clearly fails to capture the reusability of real-world secure processors.

We show how to realize UC-secure commitments assuming only transparent enclaves, denoted $\widehat{\mathcal{G}}_{\text{att}}$, when both parties have a secure processor (since otherwise the task would have been impossible as noted earlier). Although intuition is quite simple — the committer could commit the value to its local enclave, and later ask the enclave to sign the opening — it turns out that this natural protocol candidate is not known to have provable security. Our actual protocol involves non-trivial techniques to achieve equivocation when the receiver is corrupt, a technical issue that arises commonly in UC proofs.

Theorem 8 (Informal)**.** *Assume that secure key exchange protocols exist. There is a $\widehat{\mathcal{G}}_{att}$-hybrid protocol that UC-realizes $\mathcal{F}_{com}$ where $\widehat{\mathcal{G}}_{att}$ is the transparent enclave functionality.*

Challenge in achieving equivocation. We note that because the committer must commit its value b to its local enclave, extraction is trivial when the committer is corrupt. The challenge is how to equivocate when the receiver is corrupt. In this case, the simulator must first simulate for the corrupt receiver a commitment-phase message which contains a valid attestation. To do so, the simulator needs to ask its enclave to sign a dummy value — note that at this moment, the simulator does not know the committed value yet. Later, during the opening phase, the simulator learns the opening from the commitment ideal functionality $\mathcal{F}_{\text{com}}$. At this moment, the simulator

must simulate a valid opening-phase message. The simulator cannot achieve this through the normal execution path of the enclave program, and therefore we must provide a special backdoor for the simulator to program the enclave's attestation on the opened value. Furthermore, it is important that a real-world committer who is potentially corrupt cannot make use of this backdoor to equivocate on the opening.

Our idea is therefore the following: the committer's enclave program must accept a special value c for which the receiver knows a trapdoor x such that $\mathsf{owf}(x) = c$, where owf denotes a one-way function. Further, the committer's enclave must produce an attestation on the value c such that the receiver can be sure that the correct c has been accepted by the committer's enclave. Now, if the committer produces the correct trapdoor x, then the committer's enclave will allow it to equivocate on the opening. Note that in the real-world execution, the honest receiver should never disclose x, and therefore this backdoor does not harm the security for an honest receiver. However, in the simulation when the receiver is corrupt, the simulator can capture the receiver's communication with $\widehat{\mathcal{G}}_{\text{att}}$ and extract the trapdoor x. Thus the simulator is now able to program the enclave's opening after it learns the opening from the $\mathcal{F}_{\text{com}}$ ideal functionality.

More specifically, the full protocol works as follows:

- First, the receiver selects a random trapdoor x, and sends it to its local enclave. The local enclave computes $c := \mathsf{owf}(x)$ where owf denotes a one-way function, and returns (c, σ) where σ is an attestation for c.

- Next, the committer receives (c, σ) from the receiver. If the attestation verifies, it then sends to its enclave the bit b to be committed, along with the value c that is the outcome of the one-way function over the receiver's trapdoor x. The committer's secure processor now signs the c value received in acknowledgment, and the receiver must check this attestation to make sure that the committer did send the correct c to its own enclave.

- Next, during the opening phase, the committer can ask its local enclave to sign the opening of the committed value, and demonstrate the attestation to the receiver to convince him of the opening. Due to a technicality commonly referred to as "equivocation" that arises in UC proofs, the enclave's "open" entry point provides the following backdoor: if the caller provides a pair of values (x, b') such that $\mathsf{owf}(x) = c$ where c was stored earlier by the enclave, then the enclave will sign b' instead of the previously committed value b.

Non-anonymous attestation. Although most of the paper is concerned about modeling anonymous attested execution as inspired by Intel's most recent SGX [7,60] and later versions of TPM [2], some secure processors instead implement non-anonymous attestation. In non-anonymous attestation, the signature binds to the platform's identity. Typically in a real-world implementation, the manufacturer embeds a long-term signing key henceforth denoted ak in each secure processor. The manufacturer then signs a certificate for the ak using its manufacturer key msk. In formal modeling, such a certificate chain can be thought of as a signature under msk, but where the message is prefixed with the platform's identity (e.g., ak).

It is not hard to see that our $(\mathcal{G}_{\text{att}}, \mathcal{G}_{\text{acrs}})$-hybrid protocol that realizes multi-party computation with a single secure processor can easily be adapted to work for the case of non-anonymous attestation as well. However, we point out that our 2-party protocol when both have secure processors would not be secure if we directly replaced the signatures with non-anonymous ones. Intuitively, since in the case of non-anonymous attestation, attestations bind to the platform's

identity, if such signatures are transferred in the clear to remote parties, then a corrupt party can convince others of an honest party's participation in the protocol simply by demonstrating a signature from that party. In comparison, if attestations were anonymous and secure processors are omnipresent, then this would not have been an issue since the adversary could have produced such a signature on its own by asking its local secure processor.

2.8 Related Work

Trusted hardware built by architects. The architecture community have been designing and building general-purpose secure processors for several decades [7, 28, 34, 39–42, 57, 58, 60, 75, 81]. The motivation for having secure processors is to minimize the trust placed in software (including the operating system and user applications) — and this seems especially valuable since software vulnerabilities have persisted and will likely continue to persist. Several efforts have been made to commercialize trusted hardware such as TPMs [2], Arm's Trustzone [6,8], and Intel's SGX [7,60]. As mentioned earlier, many of these secure processors adopt a similar attested execution abstraction despite notable differences in architectural choices, instruction sets, threat models they defend against, etc. For example, some secure processors defend against software-only adversaries [34]; others additionally defend against physical snooping of memory buses [41, 42, 58]; the latest Intel SGX defends against restricted classes of software and physical attackers, particularly, those that do not exploit certain side channels such as timing, and do not observe page swaps or memory access patterns (or observe but discard such information). A comprehensive survey and comparison of various secure processors is beyond the scope of this paper, and we refer the reader to the recent work by Shi et al. [72] for a systematization of knowledge and comparative taxonomy.

Besides general-purpose secure processors, other forms of trusted hardware also have been built and commercialized, e.g., hardware cryptography accelerators.

Cryptographers' explorations of trusted hardware. The fact that general-purpose secure processors being built in practice have more or less converged to such an abstraction is interesting. By contrast, the cryptography community have had a somewhat different focus, typically on the minimal abstraction needed to circumvent theoretical impossibilities rather than practical performance and cost effectiveness [30, 36, 45, 48, 54]. For example, previous works showed what minimal trusted hardware abstractions are needed to realize tasks such as simulation secure program obfuscation, functional encryption, and universally composable multiparty computation — tasks known to be impossible in the plain setting. These works do not necessarily focus on practical cost effectiveness, e.g., some constructions rely on primitives such as fully homomorphic encryption [30], others require sending one or more physical hardware tokens during the protocol [45,48,50,61], thus limiting the protocol's practicality and the hardware token's global reusability. Finally, a couple recent works [50, 61] also adopt the GUC framework to model hardware tokens — however, the use of GUC in these works [50, 61] is to achieve composition when an adversary can possibly pass a hardware token from one protocol instance to another; in particular, like earlier cryptographic treatments of hardware tokens [30, 45, 54], these works [50, 61] consider the same model where the hardware tokens are passed around between parties during protocol execution, and not realistic secure processors like Intel's SGX.

Use of trusted hardware in applications. Numerous works have demonstrated how to apply trusted hardware to design secure cloud systems [13, 35, 59, 69, 70], cryptocurrency systems [82],

$\mathcal{G}_{\text{att}}[\Sigma, \mathsf{reg}]$

// initialization:
On initialize: $(\mathsf{mpk}, \mathsf{msk}) := \Sigma.\mathsf{KeyGen}(1^\lambda)$, $T = \emptyset$

// public query interface:
On receive* `getpk`() from some $\mathcal{P}$: send mpk to $\mathcal{P}$

Enclave operations

// local interface — install an enclave:
On receive* `install`(idx, prog) from some $\mathcal{P} \in \mathsf{reg}$:
if $\mathcal{P}$ is honest, assert $idx = sid$
generate nonce $eid \in \{0,1\}^\lambda$, store $T[eid, \mathcal{P}] := (idx, \mathsf{prog}, \vec{0})$, send eid to $\mathcal{P}$

// local interface — resume an enclave:
On receive* `resume`(eid, inp) from some $\mathcal{P} \in \mathsf{reg}$:
let $(idx, \mathsf{prog}, \mathsf{mem}) := T[eid, \mathcal{P}]$, abort if not found
let $(\mathsf{outp}, \mathsf{mem}) := \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[eid, \mathcal{P}] := (idx, \mathsf{prog}, \mathsf{mem})$
let $\sigma := \Sigma.\mathsf{Sig}_{\mathsf{msk}}(idx, eid, \mathsf{prog}, \mathsf{outp})$, and send (outp, σ) to $\mathcal{P}$

Figure 4: A global functionality modeling an SGX-like secure processor. (Copy of Figure 1 reproduced here for convenience.) Blue (and starred*) activation points denote *reentrant* activation points. Green activation points are executed at most once. The enclave program prog may be probabilistic and this is important for privacy-preserving applications. Enclave program outputs are included in an anonymous attestation σ. For honest parties, the functionality verifies that installed enclaves are parametrized by the session identifier sid of the current protocol instance.

collaborative data analytics applications [64], and others [14, 29, 67, 71]. Due to the lack of formal abstractions for secure processors, most of these works take an approach that ranges from heuristic security to semi-formal reasoning. We hope that our work can lay the foundations for formally correctly employing secure processors in applications.

Formal security meets realistic trusted hardware. A couple earlier works have aimed to provide formal abstractions for realistic trusted hardware [12, 73], however, they either do not support cryptographically sound reasoning [73], or do not support cryptographically sound composition in general protocol design [12].

We note that our goal of having cryptographically sound formal abstractions for trusted hardware is complementary and orthogonal to the goal of providing formally correct implementations of trusted hardware [39, 81]. In general, building formally verified *implementations* of trusted hardware — particularly, one that realizes the abstractions proposed in this paper — still remains a grand challenge of our community.

Table 1: Notations.

$\mathcal{P}$	identifier of a party (potentially equipped with trusted hardware)
$\mathcal{M}$	Hardware Manufacturer
reg	registry of machines with trusted hardware
prog	a program
inp, outp	inputs and outputs resp.
mem	a program's memory tape
eid	identifier of an enclave (random nonce)
Σ	a signature scheme

3 Formal Abstractions for Attested Execution Processors

Notations. We summarize some helpful notations in Table 1.

3.1 Overview

We model all available trusted hardware processors from a given manufacturer as a single, globally shared functionality, denoted $\mathcal{G}_{\text{att}}$.

Initialization. Upon initialization, the manufacturer $\mathcal{M}$ chooses a public verification key and a signing key pair denoted (mpk, msk), for the signature scheme Σ. Later, all attestations will be signed using msk. We will focus on *anonymous* attestation as inspired by Intel SGX [7, 60].

The registry. Our idealized trusted hardware functionality $\mathcal{G}_{\text{att}}$ is parametrized by a signature scheme Σ and a global registry `reg` which contains the list of all parties that are equipped with an attested execution processor. Only the machines in `reg` will be able to call enclave operations and produce attestations under msk.

The registry `reg` aims to abstract away the manufacturer's process of registering new machines enabled with trusted hardware. For simplicity, we model it as a *static* set. We leave it as future work to support extensions such as revocation.

Public interface. $\mathcal{G}_{\text{att}}$ provides a public interface such that any party is allowed to query and obtain the public key mpk. Essentially, this models a secure public key distribution mechanism where the manufacturer can distribute a trusted global public key to all users.

We note that instead of providing a public key, in practice Intel's SGX offers an online service called Intel Attestation Service (IAS) to allow parties to verify attestations — in light of this, we can alternatively consider that our $\mathcal{G}_{\text{att}}$ offers an additional attestation verification entry point. This detail is inconsequential to our results, since *all our protocols do not require verification of attestations inside the enclave.*

Local interfaces. A local interface describes the process where a platform $\mathcal{P}$ interacts with its local trusted processor. To do this, the platform $\mathcal{P}$ must be equipped with a processor manufactured by $\mathcal{M}$, i.e., it must appear in the registry reg. Correspondingly, when a machine $\mathcal{P}$ calls an "install" instruction to $\mathcal{G}_{\text{att}}$, $\mathcal{G}_{\text{att}}$ asserts that $\mathcal{P}$ is in the registry reg. This also models the fact that for a remote party to interact with $\mathcal{P}$'s trusted processor, all commands have to be passed through

the intermediary $\mathcal{P}$. The enclave installation process (e.g., as defined in Intel's SGX [7, 52]) is abstracted into two types of invocations in our formalism:

- *Installation.* Enclave installation establishes a software enclave with program prog, linked to some identifier idx. The functionality enforces that honest hosts provide the session identifier of the current protocol instance as idx. We discuss and justify this technical condition in more detail in Section 3.2. $\mathcal{G}_{\text{att}}$ further generates a random identifier (or nonce) eid for each installed enclave, which can later be used to identify the enclave upon resume. Finally, $\mathcal{G}_{\text{att}}$ returns the generated enclave identifier eid to the caller.

- *Stateful resume.* An installed enclave can be resumed multiple times carrying state across these invocations. Each invocation identifies the enclave to be resumed by its unique eid. The enclave program prog is then run over the given input, to produce some output (together with an updated memory mem). The enclave then signs an *attestation*, attesting to the fact that the enclave with session identifier idx and enclave identifier eid was installed with a program prog, which was then executed on some input to produce outp. Note that the program's input is not included in the software attestation. This is without loss of generality, as prog may always include its inputs as part of its outputs outp to be signed.

Remark 1. Henceforth in this paper, we often omit explicitly writing the mem portion of the enclave program, but the reader should assume that the enclave program stores internal variables in between invocations.

3.2 Modeling Choices and Discussions

The enclave identifier. Each enclave has a unique identifier denoted eid. The role of the enclave identifier eid is simply to *link* multiple attestations produced by an enclave to a unique identifier. As we will see, this linkability property is crucial in many protocols, to provide "remote parties" (i.e., parties other than the host that installed the enclave) with the insurance that they are interacting with the same installed enclave program throughout a protocol execution.

Some trusted hardware platforms such as SGX assign a new identifier to each enclave. However, the uniqueness of these identifiers is not guaranteed [33]. A better approach, actually advocated by SGX's design guidelines [1], is for the installed enclave program to generate a *cryptographic nonce* by means of a trusted random number generator. This nonce, which acts as what we call an "enclave identifier", can then be included in every attestation produced by the enclave program. In this case, we consider the enclave program wrapper that generates the nonce part of our trusted computing base (TCB), and $\mathcal{G}_{\text{att}}$ captures not only the hardware TCB but additionally the minimal software TCB needed as well.

Interactions between the environment and $\mathcal{G}_{\text{att}}$. As in the standard UC framework, we assume that the environment $\mathcal{Z}$ invokes each protocol instance with a unique *session identifier* denoted sid.

Recall that $\mathcal{G}_{\text{att}}$ is a global functionality following the GUC paradigm [22]. We assume that the environment $\mathcal{Z}$ can access $\mathcal{G}_{\text{att}}$ in the following ways:

- Acting as a corrupt party;

- Acting as an honest party but only for non-challenge protocol instances. For example, the environment $\mathcal{Z}$ can access $\mathcal{G}_{\text{att}}$ through rogue protocols running on honest parties — however, by assumption these rogue protocols must have session identifiers different from the challenge *sid*.

Because of these assumptions and also due to the way $\mathcal{G}_{\text{att}}$ is defined, we make the following "non-interference" observations:

- The environment $\mathcal{Z}$ cannot install an enclave with the challenge *sid* without going through $\mathcal{A}$;
- The environment $\mathcal{Z}$ cannot access any enclave installed by a corrupt party without going through $\mathcal{A}$;
- The environment $\mathcal{Z}$ cannot access enclaves honest parties install during the challenge protocol instance; and
- $\mathcal{A}$ cannot access any enclave $\mathcal{Z}$ installed acting as an honest party (i.e., calls that did not go through $\mathcal{A}$).

Additional assumptions. We additionally assume that honest parties always invoke $\mathcal{G}_{\text{att}}$.`install` where *idx* is set to the correct *sid* corresponding to the current protocol instance. In practice, this check could be performed by an extra software wrapper, that all honest parties (in all protocols) use to interact with their trusted hardware platform. As such, our idealized $\mathcal{G}_{\text{att}}$ functionality models not only the trusted hardware platform per se, but the full "trusted computing base" (TCB) used by honest parties. However, we allow the adversary to invoke $\mathcal{G}_{\text{att}}$.`install` on any *idx* that may not correspond to the current session identifier.

3.3 A Few Useful Observations

At this point, we make a few useful observations about our $\mathcal{G}_{\text{att}}$ functionality.

Warmup: client-server outsourcing. First, observe that the enclave program prog and all inputs inp are observable by the platform $\mathcal{P}$ that owns the secure processor, since $\mathcal{P}$ must be an intermediary in all interactions with its local secure processor.

Although at first glance, it might seem that the enclave program does not retain any secrets from $\mathcal{P}$, we point out that this is not true: specifically the enclave program prog may be randomized. It can generate a random key and perform a key exchange with a remote client. In this way, a remote client can establish a secure channel with the enclave such that the intermediary $\mathcal{P}$ cannot eavesdrop on or tamper with the communication. In fact, in Appendix B, as a warmup exercise, we formalize this simple application referred to as outsourcing, where an honest client outsources data and computation to a remote server equipped with a secure processor.

$\mathcal{G}_{\text{att}}$ does not give authenticated channels. Although $\mathcal{G}_{\text{att}}$ allows remote parties to check an attestation an be convinced that it was produced by a program running in an installed enclave, it does not allow remote parties to authenticate *each other*.

Fact 1. *The functionality $\mathcal{F}_{auth}$ cannot be realized in the $\mathcal{G}_{att}$-hybrid model, for networks of at least two parties.*

It is known that the ideal authenticated channels functionality $\mathcal{F}_{\text{auth}}$ is impossible to realize in the "plain" UC model [21]. This impossibility result, and its proof, remain valid in the $\mathcal{G}_{\text{att}}$-hybrid model, as an adversary that corrupts a party $\mathcal{P} \in \texttt{reg}$ can simulate any messages issued by another party.

4 Stateful Obfuscation from $\mathcal{G}_{\text{att}}$

As is well-known by the cryptography community, trusted hardware can allow us to circumvent known theoretical impossibilities [15, 31, 36, 37, 45, 48]. For example, virtual blackbox obfuscation is known to be impossible in a plain model with standard assumptions [11]. However, prior work show that virtual blackbox obfuscation may be realized from even *stateless* trusted hardware tokens [48].

We show that a trusted hardware functionality such as $\mathcal{G}_{\text{att}}$ allows us to realize powerful primitives that would otherwise not be possible with stateless trusted hardware (and thus would not be possible to realize with program obfuscation).

Stateful obfuscation and motivating application. To do this, we will formally define a primitive called *stateful obfuscation*. The best way to understand stateful obfuscation is to imagine the following application: suppose that a hospital has a differentially private data analytics program that makes queries over a medical database consisting of many users' records. The medical data is privacy sensitive, and moreover the data analytics algorithm is proprietary. Therefore the hospital obfuscates the program as well as the data prior to distribution (in this case, consider the union of the program and the data as the program to be obfuscated). As users make queries to the program, the privacy budget gets consumed. Therefore, the program should keep track of the remaining privacy budget and when the budget depletes, the program should output $\perp$ upon any new queries. Traditional notions of program obfuscation, even simulation-secure definitions, are unable to support this application since the obfuscated program cannot keep state, and thus is always vulnerable to a rewinding attack.

Summary of results. Informally, in this section, we will show two results:

1. Stateful obfuscation cannot be realized from stateless hardware. Since previous works have shown that one can build program obfuscation from stateless hardware, this also rules out building stateful obfuscation from standard notions of program obfuscation.
2. We show that there exists a protocol that realizes stateful obfuscation from $\mathcal{G}_{\text{att}}$.

4.1 Formal Definitions

We now formally define stateful obfuscation — see Figure 5. Our formal definition of stateful obfuscation involves a client $\mathcal{C}$ and a server $\mathcal{S}$. The client and the server may perform an interactive setup at the end of which the server $\mathcal{S}$ obtains an obfuscated version of the program. Only the setup phase can be interactive: later during the evaluation phase, the server $\mathcal{S}$ should be able to evaluate the obfuscated program on its own without the client's help — a property referred to as "non-interactive evaluation". Our notion of stateful obfuscation is also designated-receiver (in this case the server is the receiver), in the sense that the server $\mathcal{S}$ cannot pass the obfuscated program

$\mathcal{F}_{\text{statefulobf}}[sid, \mathcal{C}, \mathcal{S}]$

// obfuscate a function f:
On receive ("obfuscate", f) from $\mathcal{C}$:
  notify $\mathcal{S}$, $\mathcal{A}$ of $|f|$ and store $(f, \mathsf{st} := \bot)$
  send a public delayed output "okay" to $\mathcal{C}$

// evaluate with input x:
On receive* ("compute", x) from $\mathcal{S}$:
  assert that (f, st) is stored
  let $(y, \mathsf{st}') := f(x, \mathsf{st})$, let $\mathsf{st} := \mathsf{st}'$ and send y to $\mathcal{S}$

Figure 5: The ideal stateful obfuscation functionality.

around such that they can be evaluated by other parties. The choice of designated-receiver is in some sense inevitable for stateful obfuscation, since if the obfuscated program can be passed around in an unrestricted manner, a rewinding attack is always possible.

Definition 1 (Stateful obfuscation)**.** *A $\mathcal{G}$-hybrid protocol π between a client $\mathcal{C}$ and a server $\mathcal{S}$ is said to be a stateful obfuscation scheme if the following holds:*

- *Security. π securely realizes $\mathcal{F}_{statefulobf}$ when the client $\mathcal{C}$ is honest and the server $\mathcal{S}$ is possibly corrupt.*
- *Non-interactive evaluation. On input* inp *from the environment $\mathcal{Z}$, the protocol for $\mathcal{S}$ to evaluate* outp *may only invoke $\mathcal{S}$ and $\mathcal{G}$ but not $\mathcal{C}$.*

A functionality $\mathcal{G}$ possibly representing hardware tokens is said to be *stateless*, if there is a probabilistic polynomial-time function g, such that whenever party $\mathcal{P}$ sends message m to $\mathcal{G}$, $\mathcal{G}$ replies with $g(m, \mathcal{P})$. In other words, $\mathcal{G}$ evaluates a fixed probabilistic polynomial-time function and does not store state.

4.2 Impossibility in the Standard Model or with Stateless Tokens

Theorem 9 (Impossibility of stateful obfuscation from stateless trusted hardware)**.** *If $\mathcal{G}$ is stateless, then no $\mathcal{G}$-hybrid protocol (absent other functionalities) can realize stateful obfuscation.*

On the other hand, it is not hard to see that (designated-receiver) virtual-blackbox obfuscation can be realized from stateless trusted hardware.

Proof. Imagine that st is a counter initialized to 0 by the honest client. Let $s \xleftarrow{\$} \mathbb{Z}_p$ be a randomly chosen secret where p is a $\Omega(\lambda)$-bit prime. Let F_s denote a (n, t)-Shamir secret sharing polynomial with the threshold t and encoding the secret s. In other words, $F_s \in \mathbb{F}_p[x]$ is a random polynomial of degree $t-1$ whose 0-th coefficient encodes the secret s. Suppose prog_s is the stateful program that outputs $F_s(\mathsf{inp})$ upon the i-th invocation if $i \leq t$; and outputs $\bot$ upon further invocations. More formally, $\mathsf{prog}_s(\mathsf{st}, \mathsf{inp})$ computes the following stateful function:

If $\mathsf{st} \geq t$, output $(\mathsf{st}+1, \bot)$; else output $(\mathsf{st}+1, F_s(\mathsf{inp}))$

For the sake of contradiction, suppose there exists a stateful obfuscation scheme π in the $\mathcal{G}$-hybrid world (that does not call other functionalities). By definition, π is a $\mathcal{G}$-hybrid world protocol, such that when the environment $\mathcal{Z}$ inputs inp to $\mathcal{S}$, $\mathcal{S}$ interacts only with $\mathcal{G}$ but not $\mathcal{C}$ before outputting an answer. Suppose that in some real-world execution of π, $\mathcal{C}$ receives input prog_s from the environment $\mathcal{Z}$, where $s \xleftarrow{\$} \mathbb{Z}_p$ is chosen at random by $\mathcal{Z}$. For π to securely realize $\mathcal{F}_{\text{statefulobf}}$, when $\mathcal{S}$ first receives any input $\mathsf{inp} \in \mathbb{Z}_p$ from $\mathcal{Z}$, it performs some interactions with $\mathcal{G}$, and except with negligible probability, it outputs $F_s(\mathsf{inp})$ to $\mathcal{Z}$. Now the adversary $\mathcal{S}$ can simply rewind and rerun the honest evaluation protocol using $t+1$ different inputs $\mathsf{inp}_1, \ldots, \mathsf{inp}_{t+1}$. Since the evaluation protocol interacts only with $\mathcal{G}$ and $\mathcal{G}$ is stateless, it is not hard to see that with all but negligible probability, $\mathcal{A}$ must output to $\mathcal{Z}$ the correct $F_s(\mathsf{inp}_i)$ upon the i-th evaluation. Clearly in the real-world execution, $\mathcal{A}$ can output the secret s with all but negligible probability, whereas in an ideal-world execution, $\mathcal{A}$ cannot output s except with negligible probability. Hence the protocol π cannot realize $\mathcal{F}_{\text{statefulobf}}$. □

4.3 Construction from Attested Execution Processors

We construct a protocol $\mathbf{Prot}_{\text{statefulobf}}$ that securely realizes $\mathcal{F}_{\text{statefulobf}}$ (Figure 6). Here we make the simplifying assumption (which does not impact our previous impossibility result) that the lengths of the computed function f, its input x, and output y are of fixed and publicly known.

For concreteness and ease of exposition, we will leverage Diffie-Hellman for key exchange and an authenticated encryption scheme. It is not hard to modify our scheme for any secure key exchange protocol (possibly with more rounds). Since the existence of secure key exchange protocols imply the existence of authenticated encryption, it would suffice to assume secure key exchange for theoretical feasibility.

Most of the protocol is quite natural, namely, the client establishes a secure channel with the server's enclave, and sends the function f to be obfuscated to the server's enclave. Afterwards, the server is allowed to supply inputs to the enclave program to perform evaluation. We point out one technical subtlety: the enclave program's "compute" entry point has a backdoor y'. If the input $y' \neq \perp$, the enclave will simply output y' and its attestation. Otherwise, the enclave will output the true outcome of the evaluation along with its attestation. In the real-world protocol, an honest server will always supply a value $y' = \perp$; if a malicious server supplies $y' = \perp$, it does not learn anything more than an attestation on y'. In the simulation, however, the simulator will pick a canonical function f_0, and simulate an authenticated ciphertext for f_0. Therefore, in the simulation, the enclave program learns the function f_0 (unless simulation is aborted prematurely). Then, at some later point, the simulator will learn the outcome y from the ideal functionality $\mathcal{F}_{\text{statefulobf}}$. At this moment, the simulator must have a way to program the enclave to output the desired value y rather than $f_0(x)$. This backdoor in the enclave program's "compute" entry point provides the simulator the ability to perform such programming.

Theorem 10 (Stateful obfuscation from $\mathcal{G}_{\text{att}}$)**.** *Assume that the signature scheme Σ is existentially unforgeable under chosen message attacks, the Decisional Diffie-Hellman assumption holds in the algebraic group adopted, the authenticated encryption scheme AE is perfectly correct and satisfies the standard notions of INT-CTXT and semantic security. Then, the $\mathcal{G}_{att}$-hybrid protocol $\mathbf{Prot}_{statefulobf}$ UC-realizes $\mathcal{F}_{statefulobf}$ when the client $\mathcal{C}$ is honest, and that the server $\mathcal{S}$ is corrupt.*

Proof. We focus on the case where the server is corrupt; the case where both parties are honest is trivial as all communications occur over secret channels.

$\mathsf{prog}_{\text{statefulobf}}$

On input $(\text{“keyex”}, g^a)$: $b \leftarrow_\$ \mathbb{Z}_p$, store $\mathsf{sk} := (g^a)^b$, return (g^a, g^b)

On input $(\text{“obfuscate”}, \mathsf{ct})$:
let $f := \mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct})$, assert decryption success
store $(f, \mathsf{st} := \bot)$ and return “okay”

On input* $(\text{“compute”}, x, y')$:
assert that (f, st) is stored
if $y' \neq \bot$ return y'
else let $(y, \mathsf{st}') := f(x, \mathsf{st})$, let $\mathsf{st} := \mathsf{st}'$ and return y

$\mathbf{Prot}_{\text{statefulobf}}[sid, \mathcal{C}, \mathcal{S}]$

Server $\mathcal{S}$:

On receive $(\text{“keyex”}, g^a)$ from $\mathcal{C}$:
let $eid := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{statefulobf}})$
let $((g^a, g^b), \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“keyex”}, g^a))$ and send (eid, g^b, σ) to $\mathcal{C}$

On receive $(\text{“obfuscate”}, \mathsf{ct})$ from $\mathcal{C}$: $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“obfuscate”}, \mathsf{ct}))$

On receive* $(\text{“compute”}, x)$ from $\mathcal{Z}$: let $(y, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, x, \bot))$, output y

Client $\mathcal{C}$:

On input $(\text{“obfuscate”}, f)$ from $\mathcal{Z}$:
let $a \leftarrow_\$ \mathbb{Z}_p$, $\mathsf{mpk} := \mathcal{G}_{\text{att}}.\texttt{getpk}()$
send $(\text{“keyex”}, g^a)$ to $\mathcal{S}$, await (eid, g^b, σ) from $\mathcal{S}$
assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((sid, eid, \mathsf{prog}_{\text{statefulobf}}, (g^a, g^b)), \sigma)$ and let $\mathsf{sk} := (g^b)^a$
let $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f)$ and send $(\text{“obfuscate”}, \mathsf{ct})$ to $\mathcal{S}$

Figure 6: A protocol $\mathbf{Prot}_{\text{statefulobf}}$ that realizes the stateful obfuscation functionality $\mathcal{F}_{\text{statefulobf}}$. The public group parameters (g, p) are hardcoded into $\mathsf{prog}_{\text{statefulobf}}$.

Ideal-world simulator Sim. We first describe an ideal-world simulator Sim, and then show that no p.p.t. environment $\mathcal{Z}$ can distinguish the ideal-world and real-world executions.

- Unless noted otherwise below, any communication between $\mathcal{Z}$ and $\mathcal{A}$ or between $\mathcal{A}$ and $\mathcal{G}_{\text{att}}$ is simply forwarded by Sim.
- The simulator Sim starts by emulating the setup of a secure channel between $\mathcal{C}$ and $\mathcal{G}_{\text{att}}$. Sim sends g^a to $\mathcal{A}$ (that controls the corrupted $\mathcal{S}$) for a randomly chosen a.
- When Sim receives a tuple (eid, g^b, σ) from $\mathcal{A}$, Sim aborts outputting sig-failure if σ would be validated by an honest $\mathcal{C}$, and yet Sim has not recorded the following $\mathcal{A} \Leftrightarrow \mathcal{G}_{\text{att}}$ communication:
  - $eid := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{statefulobf}})$;
  - $((g^a, g^b), \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“keyex”}, g^a))$

  Else, Sim computes $\mathsf{sk} = g^{ab}$.

- The simulator Sim chooses a canonical function f_0 and sends $(\text{“obfuscate”}, \mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f_0))$ to $\mathcal{A}$.
- If $\mathcal{A}$ makes a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, \_, \_))$ call and Sim has not observed a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“obfuscate”}, \mathsf{ct}))$ call, where ct was the ciphertext sent previously to $\mathcal{A}$, Sim forwards the message to $\mathcal{G}_{\text{att}}$ and aborts with output $\mathsf{authenc\text{-}failure}$ if $\mathcal{G}_{\text{att}}$ returns an output other than $\perp$.
- If $\mathcal{A}$ makes a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, x, \perp))$ call and a correct “obfuscate” call was previously observed, Sim sends $(\text{“compute”}, x)$ to $\mathcal{F}_{\text{statefulobf}}$ and receives y. It then replaces the message to $\mathcal{G}_{\text{att}}$ by $(\text{“compute”}, \perp, y)$, and forwards the response (y, σ) to $\mathcal{A}$.
- If $\mathcal{A}$ makes a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, \_, y'))$ call (where $y' \neq \perp$) and a correct “obfuscate” call was previously observed, Sim simply forwards the message to $\mathcal{G}_{\text{att}}$ and returns the response (y', σ) to $\mathcal{A}$.

We now prove the indistinguishability of the real-world and ideal-world executions through a sequence of hybrids.

Claim 1. *Assume that the signature scheme Σ is secure, except with negligible probability, the simulated execution does not abort outputting* $\mathsf{sig\text{-}failure}$.

Proof. Straightforward reduction to the security of the digital signature scheme Σ. □

Hybrid 1. Identical to the simulated execution, but the secret key $\mathsf{sk} = g^{ab}$ shared between $\mathcal{C}$ and $\mathcal{G}_{\text{att}}$ is replaced with a random element from the appropriate domain.

Claim 2. *Assume that the DDH assumption holds, then Hybrid 1 is computationally indistinguishable from the simulated execution.*

Proof. Straightforward by reduction to the DDH assumption. □

Claim 3. *Assume that* AE *satisfies INT-CTXT security. It holds that in Hybrid 1,* $\mathsf{authenc\text{-}failure}$ *does not happen except with negligible probability.*

Proof. Straightforward by reduction to the INT-CTXT security of authenticated encryption. If $\mathcal{A}$ never makes a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“obfuscate”}, \mathsf{ct}))$ call where ct is the ciphertext previously sent by Sim, then with all but negligible probability, $\mathsf{prog}_{\text{statefulobf}}$ will not have stored some (f, st) and will return $\perp$ on any “compute” call. □

Hybrid 2. Instead of sending $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f_0)$ to $\mathcal{A}$, we now send $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f)$ where f is the honest client's true input.

Claim 4. *Assume that* AE *is semantically secure, Hybrid 2 is computationally indistinguishable from Hybrid 1.*

Proof. Straightforward reduction to the semantic security of authenticated encryption. □

Hybrid 3. Now instead of injecting the true output y into the "compute" message sent by $\mathcal{A}$ to $\mathcal{G}_{\text{att}}$, the message is simply forwarded to $\mathcal{G}_{\text{att}}$ and the output returned to $\mathcal{A}$.

Claim 5. *Hybrid 3 is identically distributed as Hybrid 2.*

Hybrid 4. Now instead of using a random key between $\mathcal{C}$ and $\mathcal{G}_{\text{att}}$, we switch back to using the real key g^{ab}.

Claim 6. *Assume that the DDH assumption holds, then Hybrid 4 is computationally indistinguishable from Hybrid 3.*

Proof. Straightforward by reduction to the DDH assumption. □

Finally, observe that conditioned on the simulator not aborting and AE being perfectly correct, Hybrid 4 is identically distributed as the real execution.

□

5 Composable 2-Party Computation

5.1 Lower Bound

We first consider the feasibility of realizing universally composable multi-party computation when not all parties have a secure processor. We show a negative result.

Theorem 11 (Impossibility of UC-secure MPC when not all parties have a secure processor)**.** *If at least one party $\mathcal{P}$ is not equipped with trusted hardware (i.e., $\mathcal{P} \notin \texttt{reg}$), it is impossible to UC-realize MPC in the $\mathcal{G}_{att}$-hybrid model, even with pairwise authenticated channels.*

Proof. To show that it is impossible to realize general MPC, we show that one particular functionality, namely two-party commitments, cannot be realized. We follow the same ideas as in the proof of impossibility of realizing commitments in the plain authenticated model (i.e., without $\mathcal{G}_{\text{att}}$) from [23]. We consider a commitment between two parties, the committer $\mathcal{P}_i$ and the receiver $\mathcal{P}_j$, using some protocol π. Without loss of generality, we will assume that $\mathcal{P}_i \neq \texttt{reg}$, i.e., that the committer is not equipped with trusted hardware.

The proof now proceeds identically to the one in [23]: Consider a real-world adversary $\mathcal{A}$ that corrupts the committer $\mathcal{P}_i$. This adversary is a "dummy adversary" that simply forwards messages between $\mathcal{P}_i$ and the environment $\mathcal{Z}$.

Now, $\mathcal{Z}$ picks a bit b at random and honestly follows $\mathcal{P}_i$'s part of the protocol π to commit to b. If the protocol π requires $\mathcal{P}_i$ to make any "install" or "resume" calls to $\mathcal{G}_{\text{att}}$, $\mathcal{Z}$ simply ignores those calls (as $\mathcal{P}_i \notin \texttt{reg}$). Once $\mathcal{P}_j$ acknowledges receipt of the commitment, $\mathcal{Z}$ instructs $\mathcal{A}$ to perform $\mathcal{P}_i$'s part of the protocol π to decommit to b. Again, $\mathcal{A}$ ignores any (unsuccessful) calls to $\mathcal{G}_{\text{att}}$ proscribed by π. Finally, when $\mathcal{P}_j$ outputs ("open", b'), $\mathcal{Z}$ outputs 1 if $b = b'$ and 0 otherwise.

As $\mathcal{P}_j$ outputs a receipt before the decommitment phase starts, an ideal-world simulator Sim for the pair $(\mathcal{A}, \mathcal{Z})$ must send some value b' to $\mathcal{F}_{\text{com}}$, before learning the value of the bit b. However, for the simulation to be faithful, Sim's value b' should be equal to the value b chosen by $\mathcal{Z}$, which contradicts commitment secrecy. □

$\mathsf{prog}_{\mathrm{2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, b]$

On input ("keyex"): $y \xleftarrow{\$} \mathbb{Z}_p$, return g^y

On input ("send", g^x, inp_b):
  assert that "keyex" has been called
  $\mathsf{sk} := (g^x)^y$, $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(\mathsf{inp}_b)$, return ct

On input ("compute", ct, v):
  assert that "send" has been called and ct not seen
  $\mathsf{inp}_{1-b} := \mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct})$, assert that decryption succeeds
  if $v \neq \perp$, return v; else return $\mathsf{outp} := f(\mathsf{inp}_0, \mathsf{inp}_1)$

$\mathbf{Prot}_{\mathrm{2pc}}[sid, f, \mathcal{P}_0, \mathcal{P}_1, b]$

On input inp_b from $\mathcal{Z}$:
  $eid := \mathcal{G}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathrm{2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, b])$
  henceforth denote $\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\cdot) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(eid, \cdot)$
  $(g^y, \sigma) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"keyex"})$
  send (eid, g^y, σ) to $\mathcal{P}_{1-b}$, await (eid', g^x, σ')
  assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((sid, eid', \mathsf{prog}_{\mathrm{2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, 1-b], g^x), \sigma')$
  $(\mathsf{ct}, \_) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"send"}, g^x, \mathsf{inp}_b)$, send ct to $\mathcal{P}_{1-b}$, await ct'
  $(\mathsf{outp}, \_) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"compute"}, \mathsf{ct}', \perp)$, output outp

Figure 7: Composable 2-party computation: both parties have secure processors. (Copy of Figure 2 reproduced here for the reader's convenience.) The group parameters (g, p) are hardcoded into $\mathsf{prog}_{\mathrm{2pc}}$.

Note that this lower bound proof would fail assuming all nodes have access to trusted hardware. In this case, the simulator will be able to trivially extract any communication between a corrupt party and $\mathcal{G}_{\mathrm{att}}$, as this communication cannot be emulated by the environment. This extraction capability is what gives the simulator "extra-power" over the real-world adversary. The above lower bound proof would fail since the simulator is in some sense more powerful than the real-world receiver.

5.2 Composable 2-Party Computation When Both Have Secure Processors

As a warmup exercise, we present in Figure 7 a protocol for realizing composable 2-party computation assuming that both parties have a secure processor. For concreteness and ease of exposition, our protocol makes use of Decisional Diffie-Hellman (DDH) and authenticated encryption. It is easy to see that our construction and proofs extend in the most natural manner to any secure key exchange protocol (possibly with more rounds). Further, since the existence of authenticated encryption and digital signatures is implied by that of key exchange, it suffices to assume key exchange for theoretical feasibility. For simplicity, we also assume that parties' inputs and the function output are of a fixed (a-priori known) length.

Most parts of the protocol in Figure 7 are natural. Basically, both parties' enclaves establish a secure channel and send the parties' respective inputs to the other enclave (we assume a Diffie-

Hellman key exchange for concreteness but any secure key exchange would do). The two enclaves then each perform evaluation, and each party queries its local enclave for the outcome.

However, we point out a few technicalities that arise in the construction or proof. First, the enclave program's "compute" entry point has a backdoor v. A real-world honest party always supplies a value $v = \perp$, in which case the enclave will sign the true outcome and return the attested output. However, the simulator will make use of this backdoor v program the enclave to sign outputs of its choice, when it has learned the outcome from the ideal functionality $\mathcal{F}_{\text{2pc}}$. Note that extraction is easy since the protocol requires that both parties send their respective inputs to $\mathcal{G}_{\text{att}}$, and therefore the simulator can capture the adversary's communication with $\mathcal{G}_{\text{att}}$ and extract the adversary's input. Finally, as mentioned earlier in Section 2.4, in the simulation, the simulator needs to effectively simulate the honest enclave on the adversary's secure processor — and this is possible here since the attesation is anonymous.

Theorem 12 (2-party computation from $\mathcal{G}_{\text{att}}$ when both have secure processors)**.** *Assume that the DDH assumption holds, the authenticated encryption scheme is perfectly correct, semantically secure, and INT-CTXT secure, and Σ is a secure signature scheme, the protocol described in Figure 7 UC-realizes $\mathcal{F}_{2pc}^{f}$.*

Proof. Both the protocol in Figure 7 and the proof are in some sense a subset of those for *fair* 2-party computation when both parties have secure processors (Section 7.6, Theorem 17). We therefore simply omit the proof here and refer the reader to the proof of Theorem 17. □

6 Composable Multi-Party Computation with a Single Secure Processor and an Augmented Global CRS

Earlier we showed that even when a single party does not have a secure processor, universally composable MPC is unfortunately not possible in the $\mathcal{G}_{\text{att}}$ hybrid world. In this section, we explore how to introduce minimal global setup assumptions to circumvent this impossibility.

For concreteness and ease of exposition, our protocols in this section will make use of a public-key encryption scheme (for key exchange), a non-interactive witness indistinguishable proof system, a digital signature scheme, and authenticated encryption. It is not hard to observe that our protocol and proofs extend naturally to any secure key exchange protocol (possibly with more rounds), and any interactive witness indistinguishable proof system. Since the existence of secure key exchange implies the existence of interactive witness indistinguishable proof systems, authenticated encryption, and digital signatures, secure key exchange protocols suffice for theoretical feasibility.

6.1 Augmented Global CRS

We will leverage the $\mathcal{G}_{\text{acrs}}$ global functionality described in Figure 8. $\mathcal{G}_{\text{acrs}}$ was first proposed by Canetti et al. [22].

We briefly explain $\mathcal{G}_{\text{acrs}}$, also referred to as an augmented global common reference string. In particular, $\mathcal{G}_{\text{acrs}}$ provides a public common reference string that is honestly generated. Honest parties never have to query $\mathcal{G}_{\text{acrs}}$ for any additional information — in this sense $\mathcal{G}_{\text{acrs}}$ requires only minimal additions atop a standard global common reference string. On the other hand, $\mathcal{G}_{\text{acrs}}$ leaves a backdoor for the adversary, such that the adversary can obtain identity keys pertaining to their party identifiers. Later, our protocol will demonstrate how the simulator can leverage

$\mathcal{G}_{\text{acrs}}$

On initialize: $(\mathsf{epk}, \mathsf{esk}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$, $(\mathsf{ssk}, \mathsf{vk}) \xleftarrow{\$} \Sigma.\mathsf{Gen}(1^\lambda)$, $\mathsf{crs} \leftarrow \mathsf{NIWI.Gen}(1^\lambda)$
On receive* "crs" from $\mathcal{P}$: return $\mathcal{G}_{\text{acrs}}.\mathsf{mpk} := (\mathsf{epk}, \mathsf{vk}, \mathsf{crs})$
On receive* "idk" from $\mathcal{P}$: assert $\mathcal{P}$ is corrupt, and return $\Sigma.\mathsf{Sign}_{\mathsf{ssk}}(\mathcal{P})$

Figure 8: Global augmented common reference string. Generates a public encryption key pair, a signing key pair, and a common reference string for the witness indistinguishable proof system. Upon query from a (corrupt) party, returns a signature on the party's identifier henceforth called the *identity key*.

corrupt parties' identity keys to perform extraction and equivocation, two key elements of protocol composition proofs.

More concretely, a party's identity key is a signature of its party identifier under a master signing key. This signature can be verified with $\mathcal{G}_{\text{acrs}}.\mathsf{vk}$. Henceforth we use the following notation

$$\mathsf{check}(\mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{P}, \mathsf{idk})$$

to denote the following: first, parse $\mathcal{G}_{\text{acrs}}.\mathsf{mpk} := (\_, \mathsf{vk}, \_)$; next call $\Sigma.\mathsf{Ver}_{\mathsf{vk}}(\mathcal{P}, \mathsf{idk})$. In other words, use the signature verification key inside $\mathcal{G}_{\text{acrs}}.\mathsf{mpk}$ to verify whether idk is a valid signature on the party identifier $\mathcal{P}$.

Additionally, $\mathcal{G}_{\text{acrs}}$ also generates a public encryption key denoted epk, and a global common reference string crs for the proof system that we will adopt.

6.2 NP Languages Adopted in the Protocol

Our protocol relies on witness indistinguishable proofs whose formal definitions are presented in Appendix A.4. For ease of exposition, we define non-interactive witness indistinguishable proofs with a global common reference string — but it is not hard to see that our protocol and proofs naturally extend to interactive witness indistinguishable proofs (without a common reference string).

We define the NP language we will rely on and related shorthands.

Language for proving signatures from secure processors. Let a statement be of the form $\mathsf{stmt} := (sid, eid, C, \mathsf{mpk}, \mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{P}, \mathsf{msg})$ where $\mathcal{G}_{\text{acrs}}.\mathsf{mpk} := (\mathsf{epk}, \mathsf{vk}, \mathsf{crs})$ and a witness be of the form $w := (r, \sigma, \mathsf{idk}[\mathcal{P}])$. The NP relation is defined as below:

$$\exists(\mathsf{msg}, r, \sigma, \mathsf{idk}[\mathcal{P}]) \text{ s.t. } C = \mathsf{PKE.Enc}_{\mathsf{epk}}((\sigma, \mathsf{idk}[\mathcal{P}]), r) \text{ and}$$
$$\big(\mathsf{Ver}_{\mathsf{mpk}}(sid, eid, \mathsf{prog}_{\text{mpc}}[f, \mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n], \mathsf{msg}, \sigma) \text{ or } \mathsf{check}(\mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{P}, \mathsf{idk}[\mathcal{P}]) = 1\big)$$

More informally, the statement basically asserts that the plaintext encrypted under C either contains a valid signature σ for the message msg signed by a secure processor, or it contains $\mathcal{P}$'s identity key $\mathsf{idk}[\mathcal{P}]$.

Abbreviated notations for zero-knowledge proofs. Henceforth we will use the following shorthand, omitting public parameters that are implicit from the context. We define

$$\mathsf{NIWI.Prove}((\mathcal{P}, \mathsf{msg}, C), (r, \sigma, \mathsf{idk}[\mathcal{P}])) :=$$
$$\mathsf{NIWI.Prove}(\mathsf{crs}, (sid, eid, C, \mathsf{mpk}, \mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{P}, \mathsf{msg}), (r, \sigma, \mathsf{idk}[\mathcal{P}]))$$

Further, given $(\mathcal{P}, \mathsf{msg}, \sigma)$, we define the subroutine $\psi(\mathcal{P}, \mathsf{msg}, \sigma)$ as follows:

- Generate r at random, let $C := \mathsf{PKE.Enc}_{\mathsf{epk}}((\sigma, \perp), r)$.
- Let $\pi := \mathsf{NIWI.Prove}((\mathcal{P}, \mathsf{msg}, C), (r, \sigma, \perp))$.
- We now define

$$\psi(\mathcal{P}, \mathsf{msg}, \sigma) := (\mathsf{msg}, C, \pi)$$

In other words, $\psi(\mathcal{P}, \mathsf{msg}, \sigma)$ transforms an attestation σ to a witness-indistinguishable proof, under appropriate public parameters, of the fact that either C encrypts a valid signature on msg, or it encrypts $\mathcal{P}$'s identity key.

6.3 Detailed Protocol

We present our detailed protocol in Figure 9. The key insight behind the protocol is that the enclave program is parametrized by $\mathcal{G}_{\text{acrs}}.\mathsf{mpk}$. In the simulation, the simulator will obtain corrupt parties' identity keys from $\mathcal{G}_{\text{acrs}}.\mathsf{mpk}$. We embed backdoors in the enclave program named "extract" and "program", such that when the simulator provides corrupt parties' identity keys, the enclave program will 1) leak to the simulator corrupt parties' secret keys for their respective secure channel with the enclave, allowing the simulator to extract corrupt parties' inputs; and 2) let the simulator program corrupt parties' outputs. We stress that these backdoors do not harm honest parties' security because honest parties never even query $\mathcal{G}_{\text{acrs}}$ for their identity keys.

Otherwise, the protocol proceeds in the a natural manner (but with subtleties), where each party encrypts its input to the server's enclave, additionally each party encrypts a a secret session key later used to form a secure channel with the enclave. Through an attestation sent in the response, each party verifies that the enclave has correctly registered their respective input. If this is indeed the case, all parties acknowledge the enclave's eid to each other over a pairwise secure channel. If all parties confirm that they are talking to the same enclave, they then send "ok" to the enclave over a secure channel. Upon collecting "ok" messages from all parties, the enclave proceeds with the evaluation, and finally, signs and returns the output. One subtlety is that all parties must acknowledge that they are talking to the same enclave over pairwise secure channels before the enclave can proceed with the evaluation — since otherwise the adversary can simply impersonate one of the parties and supply a malicious input on behalf of the victim.

Again, for sake of simplicity, we will assume that all parties' inputs as well as the computed output are of some fixed predetermined length.

Theorem 13 (MPC from $\mathcal{G}_{\text{att}}$ when a single party has a secure processor)**.** *Assume that the signature scheme is secure, the public-key encryption is semantically secure and perfectly correct, the proof system satisfies computational soundness and witness indistinguishability, and that the authenticated encryption scheme is perfectly correct, semantically secure, and INT-CTXT secure, then the protocol described in Figure 9 UC-realizes* $\mathcal{F}_{mpc}^{f}$.

Proof. We now prove the above theorem. Henceforth, we say that the tuple $\psi := (\mathsf{msg}, C, \pi)$ verifies w.r.t. $\mathcal{P}$ and eid, iff the following holds (where other parameters in the statement, such as sid, $\mathcal{G}_{\text{acrs}}.\mathsf{mpk}$, mpk are clear from the context)

$$\mathsf{NIWI.Ver}(\mathsf{crs}, (sid, eid, C, \mathsf{mpk}, \mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{P}, \mathsf{msg})) = 1$$

$\mathsf{prog}_{\mathrm{mpc}}[f, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n]$

On input ("init"): for $i \in [n]$: $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$; return $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$

On input ("input", $\{\mathsf{ct}_i\}_{i\in[n]}$):
for $i \in [n]$: $(\mathsf{inp}_i, k_i) := \mathsf{PKE.Dec}_{\mathsf{sk}_i}(\mathsf{ct}_i)$; return $\Omega := \{\mathsf{ct}_i\}_{i\in[n]}$

On input ("extract", $\{\mathsf{idk}_i\}_{i\in[n]}$):
for $i \in [n]$: if $\mathsf{check}(\mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{P}_i, \mathsf{idk}) = 1$, $v_i := \mathsf{sk}_i$, else $v_i := \bot$; return $\{v_i\}_{i\in[n]}$

On input ("program", $\{\mathsf{idk}_i, u_i\}_{i\in[n]}$):
for $i \in [n]$: if $\mathsf{check}(\mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{P}_i, \mathsf{idk}) = 1$, $\mathsf{outp}_i := u_i$

On input ("proceed", $\{\mathsf{ct}'_i\}_{i\in[n]}$):
for $i \in [n]$: assert $\mathsf{AE.Dec}_{k_i}(\mathsf{ct}'_i) =$ "ok"
$\mathsf{outp}^* := f(\mathsf{inp}_1, \ldots, \mathsf{inp}_n)$, return "done"

On input* ("output", $\mathcal{P}_i$):
assert outp^* has been stored
if outp_i has been stored, $\mathsf{ct} := \mathsf{Enc}_{k_i}(\mathsf{outp}_i)$, else $\mathsf{ct} := \mathsf{Enc}_{k_i}(\mathsf{outp}^*)$
return ct

$\mathbf{Prot}_{\mathrm{mpc}}[sid, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, f, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n]$

Server $\mathcal{S}$:

let $eid := \mathcal{G}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathrm{mpc}}[f, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{S}, \mathcal{P}_1, \ldots, \mathcal{P}_n])$
henceforth let $\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\cdot) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(eid, \cdot)$
let $(\{\mathsf{pk}_i\}_{i\in[n]}, \sigma) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"init"})$, send $(eid, \psi(\mathcal{P}_i, \{\mathsf{pk}_i\}_{i\in[n]}, \sigma))$ to each $\mathcal{P}_i$

for each $\mathcal{P}_i$: await ("input", ct_i) from $\mathcal{P}_i$
$(\Omega, \sigma) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"input"}, \{\mathsf{ct}_i\}_{i\in[n]})$, send $\psi(\mathcal{P}_i, \Omega, \sigma)$ to each $\mathcal{P}_i$

for each $\mathcal{P}_i$: await ("proceed", ct'_i) from $\mathcal{P}_i$
$\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"proceed"}, \{\mathsf{ct}'_i\}_{i\in[n]})$
for each $\mathcal{P}_i$: $(\mathsf{ct}_i, \sigma_i) := \mathcal{G}_{\mathrm{att}}.\mathtt{resume}(\text{"output"}, \mathcal{P}_i)$, send ct_i to $\mathcal{P}_i$

Remote Party $\mathcal{P}_i$: On input inp from $\mathcal{Z}$:

await (eid, ψ) from $\mathcal{S}$
// Henceforth for $\widetilde{\psi} := (\mathsf{msg}, C, \pi)$, let $\mathsf{Ver}(\widetilde{\psi}) := \mathsf{Ver}(\mathsf{crs}, (sid, eid, C, \mathsf{mpk}, \mathcal{G}_{\mathrm{acrs}}.\mathsf{mpk}, \mathcal{P}_i, \mathsf{msg}), \pi)$
assert $\mathsf{Ver}(\psi)$, parse $\psi := (\{\mathsf{pk}_i\}_{i\in[n]}, \_, \_)$
$k \leftarrow \{0,1\}^\lambda$, $\mathsf{ct} = \mathsf{PKE.Enc}_{\mathsf{pk}}(\mathsf{inp}, k)$ where $\mathsf{pk} := \mathsf{pk}_i$
send ("input", ct) to $\mathcal{S}$, await ψ from $\mathcal{S}$, assert $\mathsf{Ver}(\psi)$, parse $\psi := (\Omega, \_, \_)$
assert $\Omega[i] = \mathsf{ct}$, send eid to all parties, wait for all parties to ack the same eid
let $\mathsf{ct}' := \mathsf{AE.Enc}_k(\text{"ok"})$, send ("proceed", ct') to $\mathcal{S}$, await ct, assert ct not seen
$\mathsf{outp} := \mathsf{Dec}_k(\mathsf{ct})$, assert ct decryption successful, return outp

Figure 9: Composable multi-party computation with a single secure processor. (Copy of Figure 3 reproduced here for convenience.) $\psi(\mathcal{P}, \mathsf{msg}, \sigma)$ outputs a tuple (msg, C, π), where π is a witness-indistinguishable proof that the ciphertext C either encrypts a valid attestation σ on msg, or encrypts $\mathcal{P}$'s identity key. PKE and AE denote public-key encryption and authenticated encryption respectively. The notation send denotes messages sent over a secure channel.

Server and some remote parties are corrupt. We construct a simulator as below.

- Unless otherwise noted, Sim passes through interactions between $\mathcal{A}$ and $\mathcal{G}_{\text{att}}$, between $\mathcal{A}$ and $\mathcal{G}_{\text{acrs}}$, and interactions between $\mathcal{A}$ and $\mathcal{Z}$.
- Sim requests $\mathcal{G}_{\text{acrs}}$ for all corrupt parties' identity keys.
- For each honest $\mathcal{P}_i$, do the following: Sim awaits (eid_i, ψ) from $\mathcal{A}$. At this moment eid_i is referred to as the challenge eid w.r.t. $\mathcal{P}_i$. Note that at this moment, different honest parties may perceive a different challenge eid. If $\psi := (\{\mathsf{pk}_i\}_{i\in[n]}, C, \pi)$ verifies w.r.t. $\mathcal{P}_i$ and eid_i but $\mathcal{G}_{\text{att}}$ did not return $(\{\mathsf{pk}_i\}_{i\in[n]}, \_)$ earlier upon a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid_i, \text{“init”})$ call, either from $\mathcal{Z}$ or $\mathcal{A}$, abort outputting $\mathsf{sig\text{-}failure}$.
- For each honest party $\mathcal{P}_i$, the simulator Sim uses the input $\mathsf{inp} = \vec{0}$, chooses k_i at random, and honestly computes the ciphertext $\mathsf{ct}_i := \mathsf{Enc}_{\mathsf{pk}_i}(\vec{0}, k_i)$, and sends the tuple ct_i to $\mathcal{A}$ (acting as the corrupt server), where pk_i is extracted from the $\{\mathsf{pk}_i\}_{i\in[n]}$ set that the simulator has received on behalf of honest $\mathcal{P}_i$ earlier.
- When $\mathcal{A}$ sends $\psi := (\Omega, C, \pi)$ for each honest $\mathcal{P}_i$: if ψ verifies w.r.t. $\mathcal{P}_i$ and eid_i, but Sim did not observe $(\Omega, \_)$ as the outcome of a prior $\mathcal{G}_{\text{att}}.\texttt{resume}(eid_i, \text{“input”}, \_)$ call by either $\mathcal{Z}$ or $\mathcal{A}$, where eid_i denotes the challenge eid from i's perspective, abort outputting $\mathsf{sig\text{-}failure}$. Also abort if $\Omega[i] \neq \mathsf{ct}_i$, where ct_i was what Sim sent to $\mathcal{A}$ earlier on behalf of $\mathcal{P}_i$.

  Once an honest party $\mathcal{P}_i$ receives such a valid message $\psi := (\Omega, C, \pi)$ from $\mathcal{A}$, Sim acks eid_i on behalf of $\mathcal{P}_i$ to each corrupt party (sent to $\mathcal{A}$).
- Sim awaits acks on eid from every corrupt party. Abort if the acks are different or if earlier, honest parties perceived different eids.
- Now the challenge eid is uniquely defined if the simulator did not abort. Henceforth whenever we write eid it refers to the unique challenge eid unless otherwise noted.

  Now, we know that $(\Omega, \_)$ is the outcome of a previous $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“input”}, \_)$ call observed by Sim. Note that since the enclave program's “input” entry is non-reentrant, if the simulation did not abort, it must be the case that all honest parties received the same Ω.

  Sim now calls $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“extract”}, \{\mathsf{idk}_i\}_{i\in[n]})$ where for each corrupt party $\mathcal{P}_i$, idk_i is set to the identity key Sim obtained from $\mathcal{G}_{\text{acrs}}$ earlier, and for each honest party, it is set to $\perp$. Sim now obtains from $\mathcal{G}_{\text{att}}$ the secret key sk_i for each corrupt $\mathcal{P}_i$. The simulator Sim can therefore decrypt the input inp_i and the session key k_i for each corrupt party $\mathcal{P}_i$. Sim sends all corrupt parties' inputs to $\mathcal{F}^f_{\text{mpc}}$ if it has not already done so. Sim obtains the output outp^* from $\mathcal{F}^f_{\text{mpc}}$.

  Sim now calls $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“program”}, \{\mathsf{idk}_i, u_i\}_{i\in[n]})$ where for each corrupt party $\mathcal{P}_i$, idk_i is set to the identity key Sim obtained from $\mathcal{G}_{\text{acrs}}$ earlier, and $u_i := \mathsf{outp}^*$; and for each honest party, the pair is set to $(\perp, \perp)$.
- On behalf of each honest party $\mathcal{P}_i$, the simulator Sim encrypts $\mathsf{ct}'_i := \mathsf{Enc}_{k_i}(\text{“ok”})$ and sends $(\text{“proceed”}, \mathsf{ct}'_i)$ to $\mathcal{A}$ (acting as the corrupt server $\mathcal{S}$).
- If $\mathcal{A}$ or $\mathcal{Z}$ ever sends $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“proceed”}, \Omega)$ for the challenge eid, the simulator checks to see for every honest $\mathcal{P}_i$, where the ct'_i the simulator has sent $\mathcal{A}$ is correctly contained in Ω. If not correctly contained but the $\mathcal{G}_{\text{att}}.\texttt{resume}$ call did not return $\perp$, abort outputting $\mathsf{authenc\text{-}failure}$. Else return the result of the $\mathcal{G}_{\text{att}}.\texttt{resume}$ call.

- The simulator passes through any call from $\mathcal{A}$ of the form $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, i)$. Notice that if the simulation did not abort and assuming æ is perfectly correct, then whenever $\mathcal{A}$ calls $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, i)$ for an honest $\mathcal{P}_i$, the returned (attested) result corresponds to an encryption of a wrong result where honest parties' inputs are $\vec{0}$. Whenever $\mathcal{A}$ calls $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, i)$ for a corrupt part $\mathcal{P}_i$, the returned (attested) result corresponds to an encryption of outp^* which was returned by the ideal functionality $\mathcal{F}_{\text{mpc}}^f$.
- For each honest $\mathcal{P}_i$, Sim awaits ct from $\mathcal{A}$ and aborts if ct was seen before. If ct successfully decrypts but Sim did not observe $(\mathsf{ct}, \_)$ as the outcome of a prior $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, i)$ query for the challenge eid, abort outputting $\mathsf{authenc\text{-}failure}$. Otherwise, Sim requests that $\mathcal{F}_{\text{mpc}}^f$ sends output to party $\mathcal{P}_i$.
- Any time during the simulation, if $\mathcal{A}$ or $\mathcal{Z}$ calls $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“program”}, \_)$ or $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“extract”}, \_)$ and provided a valid idk_i for an honest $\mathcal{P}_i$, abort outputting $\mathsf{idk\text{-}failure}$.

Lemma 1. *Assume that the signature scheme is secure, the* PKE *encryption is perfectly correct, and that* NIWI *is computationally sound. Then, the simulation does not abort with* sig-failure *except with negligible probability.*

Proof. If the simulation aborted with $\mathsf{sig\text{-}failure}$ with non-negligible probability, we can leverage the union of $\mathcal{A}, \mathcal{Z}, \mathsf{Sim}, \mathcal{G}_{\text{att}}, \mathcal{G}_{\text{acrs}}$ to build a reduction Re that breaks either the computational soundness of NIWI, or the security of the signature scheme, assuming that PKE is perfectly correct.

The idea is for Re to generate epk and store esk which is part of the global common reference string. Now Re runs the experiment with $(\mathcal{A}, \mathcal{Z})$, and waits till $\mathsf{sig\text{-}failure}$ happens — suppose it happens on the tuple (msg, C, π). At this moment, Re decrypts C with esk, and obtains a witness $(\sigma, \mathsf{idk}[\mathcal{P}_i])$ where $\mathcal{P}_i$ denotes some honest party. Now there are the following cases:

- If the witness $(\sigma, \mathsf{idk}[\mathcal{P}_i])$ is not a valid witness, Re has broken the computational soundness of NIWI.
- Else if $(\sigma, \mathsf{idk}[\mathcal{P}_i])$ is a valid witness, then we know that either σ is a valid signature on msg, or $\mathsf{idk}[\mathcal{P}_i]$ is a valid identity key for honest party $\mathsf{idk}[\mathcal{P}_i]$. In either case, the reduction can forge a new signature on a new message.

□

Claim 7. *Assume that the signature scheme is secure, then, the simulation does not abort with* idk-failure *except with negligible probability.*

Proof. By straightforward reduction to signature security. □

We prove computational indistinguishability of the real-world and simulated executions through a sequence of hybrids. By repartitioning of algorithms, let us now start treating the union of the honest parties, the ideal functionality $\mathcal{F}_{\text{mpc}}^f$, the global functionalities $\mathcal{G}_{\text{att}}$, $\mathcal{G}_{\text{acrs}}$, and the simulator Sim together as one Turing Machine.

Hybrid 1. Almost identical to the simulated execution except the following modifications:

> Every time the simulator Sim needs to compute a PKE ciphertext ct_i on behalf of an honest party $\mathcal{P}_i$, instead of encrypting the real authenticated encryption key k, Sim instead encrypts the authenticated encryption key 0. However, note that the real authenticated encryption key is still used elsewhere by the simulator and the enclave program.

Claim 8. *Assume that the public-key encryption scheme* PKE *is semantically secure, then Hybrid 1 is computationally indistinguishable from the simulated execution.*

Proof. By straightforward reduction to the semantic security of PKE. More specifically, we can have a sequence of internal hybrids, where the simulator replaces the each honest party's ct_i (referred to as the challenge ciphertext) one by one with the ciphertext obtained from a PKE challenger. For the challenge coordinate i, the simulator also uses the public key returned by the PKE challenger. For any non-challenge ciphertext ct_j, the simulator simply picks the encryption key pair itself without interacting with the PKE challenger. □

Claim 9. *Assume that* AE *satisfies INT-CTXT security. Then, in Hybrid 2,* $\mathsf{authenc\text{-}failure}$ *does not happen except with negligible probability.*

Proof. By straightforward reduction to the INT-CTXT game of AE. Observe that earlier in the hybrid sequence, the public-key ciphertext no longer encrypts the authenticated encryption keys, therefore no information is leaked to $(\mathcal{A}, \mathcal{Z})$ about the honest parties' authenticated encryption keys. □

Hybrid 2. Hybrid 2 is almost identical to Hybrid 1, except the following change:

> During a "proceed" call, if all parties authenticated encryption ciphertexts successfully decrypt to "ok", the enclave program will set outp^* to be the value returned earlier by $\mathcal{F}_{\mathrm{mpc}}^{f}$.

Notice that in Hybrid 2, whenever $\mathcal{A}$ calls $\mathcal{G}_{\mathrm{att}}.\mathtt{resume}(eid, (\text{"output"}, i))$ for an honest party $\mathcal{P}_i$, the returned result will be an encryption of outp^* instead of the wrong result computed by assuming honest parties' inputs are $\vec{0}$ that earlier hybrid used.

Claim 10. *Assume that* AE *is semantically secure. Then Hybrid 1 and Hybrid 2 are computationally indistinguishable.*

Proof. Recall that honest parties' authenticated encryption keys are not leaked to the adversary since earlier we replaced the PKE to encrypt a $\vec{0}$ key instead. Therefore, the claim holds by a straightforward reduction to the semantic security of AE.

□

Hybrid 3. Almost identical to Hybrid 2 except the following changes: for each honest party, instead of encrypting $(\vec{0}, \vec{0})$ with PKE, where the first $\vec{0}$ corresponds to the honest party's input, and the second $\vec{0}$ corresponds to the honest party's authenticated encryption key, the simulator Sim now encrypts the honest parties' true inputs, and real authenticated encryption keys (that are chosen at random by the simulator).

Claim 11. *Assume that* PKE *is semantically secure. Then Hybrid 3 and Hybrid 2 are computationally indistinguishable.*

Proof. By straightforward reduction to the semantic security of PKE. More specifically, we can build a sequence of internal hybrids where the simulator replaces honest parties' PKE ciphertexts ct_i one by one. The challenge ciphertext to be replaced is obtained from a PKE semantic security challenger. The challenge coordinate's public key pk_i also comes from the PKE challenger. All other non-challenge coordinate's public keys are generated by the simulator itself. □

Claim 12. *Assume that* PKE *is perfectly correct. Then, conditioned on simulation not aborting, Hybrid 3 is identically distributed as the real-world execution.*

Proof. Conditioned on the simulation not aborting, observe that the only difference between Hybrid 3 and the real-world execution is that in Hybrid 3, the outp^* sent to $\mathcal{A}$ is computed by $\mathcal{F}_{\text{mpc}}^f$ by evaluating f over honest parties' true inputs and what Sim extracted of corrupt parties' inputs which are decrypted from PKE ciphertexts by the enclave. In the real-world execution, the output sent to $\mathcal{A}$ is computed by the enclave program through evaluting f on the honest parties' decrypted inputs and corrupt parties' decrypted inputs (both through decryption PKE ciphertexts). It is not hard to see that if PKE is perfectly correct, then the two methods result in the same output. □

Some remote parties are corrupt but server is honest. We construct the following simulation. Throughout the simulation Sim simulates $\mathcal{G}_{\text{att}}$, except for the signing since Sim does not have $\mathcal{G}_{\text{att}}$'s signing key. However, since Sim knows the identity keys of corrupt parties, it can use the identity keys as an alternative witness when replying to $\mathcal{A}$ with a zero-knowledge proof.

- Unless otherwise noted, Sim passes through interactions between $\mathcal{A}$ and $\mathcal{G}_{\text{acrs}}$, and interactions between $\mathcal{A}$ and $\mathcal{Z}$. In this case, since the corrupt parties do not have a secure processor, we do not have to consider interactions between $\mathcal{A}$ and $\mathcal{G}_{\text{att}}$.
- Sim requests $\mathcal{G}_{\text{acrs}}$ for all corrupt parties' identity keys.
- Sim generates a random eid. Sim simulates the enclave program's "init" entry point, and generates $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}$. For every corrupt party $\mathcal{P}_i$, Sim now constructs a ciphertext C and a zero-knowledge proof denoted π vouching for $\{\mathsf{pk}_i\}_{i\in[n]}$, but using $\mathcal{P}_i$'s identity key as the witness. Sim now sends $(eid, \{\mathsf{pk}_i\}_{i\in[n]}, C, \pi)$ to $\mathcal{A}$.
- Sim now waits to receive a tuple $(\text{"input"}, \mathsf{ct})$ from each corrupt party $\mathcal{P}_i$ controlled by $\mathcal{A}$. Sim simulates the enclave program's "input" function, as well as the honest parties' inputs to the "input" function.

  Since Sim knows sk_i, it can decrypt all corrupt parties' inputs and send them to $\mathcal{F}_{\text{mpc}}^f$. Sim obtains the outcome outp^* from $\mathcal{F}_{\text{mpc}}^f$.

  Again, Sim now constructs a ciphertext C and a zero-knowledge proof π vouching for the outcome of this computation denoted Ω, but using $\mathcal{P}_i$'s identity key as the witness. Sim now sends (Ω, C, π) to $\mathcal{A}$.
- Sim acts as each honest party and acks eid to each corrupt party. Sim waits to collect corrupt parties' eid acks, and aborts if the acks are inconsistent.
- Sim now awaits $(\text{"proceed"}, \mathsf{ct}_i')$ from each corrupt party $\mathcal{P}_i$. Sim now simulates the enclave program's "proceed" function, as well as the honest parties' inputs to the "proceed" function. If the "proceed" function did not abort, Sim encrypts $\mathsf{ct} := \mathsf{Enc}_{k_i}(\mathsf{outp}^*)$ and sends ct to $\mathcal{A}$.

- If $\mathcal{A}$ allows $\mathcal{S}$'s messages to be delivered to an honest party $\mathcal{P}_i$ (recall that communication channels are UC-secure channels), Sim tells $\mathcal{F}_{\text{mpc}}^{f}$ to release outcome to $\mathcal{P}_i$.

We now show that the simulated execution and the real execution are computationally indistinguishable. During the hybrid sequences, it helps to think of the union of Sim, $\mathcal{G}_{\text{att}}$, $\mathcal{G}_{\text{acrs}}$ as a single Turing Machine (equivalent w.r.t. repartioning of algorithm boundaries).

Hybrid 0. Instead of using the corrupt parties' identity keys to construct NIWI proofs, Sim now uses the $\mathcal{G}_{\text{att}}$'s signing key to sign a signature, and then use the signature as the witness.

Claim 13. *Assume that* NIWI *is computationally witness indistinguishable, then Hybrid 0 is computationally indistinguishable from the simulated execution.*

Proof. By straightforward reduction to witness indistinguishability of the NIWI. □

Claim 14. *Assume that* PKE *is perfectly correct, then Hybrid 0 is computationally indistinguishable from the real execution.*

Proof. For every execution trace view, the real-world exeuction and Hybrid 0 can only differ if PKE decryption did not decrypt to an honest party's plaintext in which case $\mathcal{F}_{\text{mpc}}^{f}$ has a different view of honest parties' inputs than the enclave program. This cannot happen due to the perfect correctness of PKE. □

□

7 Fair 2-Party Computation

Attested execution processors often provide a trusted clock to applications. In this section, we explore the expressive power of such a trusted clock in the context of fair multi-party computation. Throughout this section, we adopt the following conventions:

- Whenever stating lower bound results, we assume that a sequential composition [20] notion of security is adopted (rather than universally composable [21, 22, 26]). Note that assuming a weaker security notion in the lower bound context makes the lower bound stronger.
- When describing our new fairness constructions, we will adopt the stronger, universally composable notion of security [21].

We formally define these notions in Appendix A.2.

7.1 Background on Fair 2-Party Computation

We first quickly review the known results about fairness in the standard model:

- A well-known lower bound by Cleve [32] shows that it is impossible to achieve fair 2-party coin flipping in the standard model. Cleve [32] also extends his impossibility result to the multi-party case, showing that fair coin toss is impossible if at least half of the parties are corrupt. Cleve's result implies that fair multi-party computation is impossible for general functionalities when half of the parties may be corrupt.

- A sequence of recent works [9, 46, 47] show that the prior folklore interpretation of Cleve's impossibility is incorrect. Specifically, the general fairness impossibility does not imply that fairness is impossible for every function. These works then make an effort at characterizing exactly which class of functions can be computed fairly [9, 46, 47].

We will now explore how a trusted clock in an attested execution processor can help with fairness.

7.2 Modeling a Trusted Clock

We assume a synchronous execution model, where protocol execution proceeds in atomic time steps called *rounds*. We assume that the trusted clocks of attested execution processors and the network rounds advance at the same rate. It is easy to adapt our model and results if the processors' trusted clocks and the network rounds do not advance at the same rate.

Execution model. For clarity, we explicitly state the execution model.

- In each round, the environment $\mathcal{Z}$ must activate each party one by one, and therefore, all parties can naturally keep track of the current round number.
- A party can perform any fixed polynomial (in λ) amount of computation when activated, and send messages.
- We consider a synchronous communication model where messages sent by an honest party will be delivered at the beginning of the next round. Whenever a party is activated in a round, it can read a buffer of incoming messages to receive messages sent to itself in the previous round.

Clock-aware functionalities. To model trusted clocks in attested execution processors, we will provide a special instruction such that enclave programs as well as ideal functionalities can query the current round number.

We say that a functionality $\mathcal{F}$ is *clock-aware* if the functionality queries the local time; otherwise we say that the functionality $\mathcal{F}$ is *clock-oblivious*.

Henceforth in this section, all our upper bound results require only *relative* clocks — in other words, all enclaves' trusted clocks need not be synchronized, since our protocol will only make use of the number of rounds that have elapsed since initialization. Therefore, we will assume the following notational conventions:

- When a functionality reads the clock, a relative round number since the first invocation of the functionality is returned;
- When an enclave program reads the clock, a relative round number since the first invocation of the enclave program is returned.

7.3 Definition: Protocols and Fairness in the Clock Model

We now give a few basic definitions for secure multi-party computation in the clock model of execution. Most importantly, we will define a notion of Δ-fairness in the clock model: roughly speaking we say that a protocol Δ-realizes some functionality, if there exists a fixed polynomial

Δ-fair 2-PC functionality $\mathcal{F}^{f,\Delta}[sid, \mathcal{P}_0, \mathcal{P}_1]$

On receive ("compute", inp_i) from $\mathcal{P}_i$ where $i \in \{0, 1\}$:
  if $\mathcal{P}_{1-i}$ has sent ("compute", inp_{1-i}): let $(\mathsf{outp}_0, \mathsf{outp}_1) := f(\mathsf{inp}_0, \mathsf{inp}_1)$

On receive ("output", δ^*) from $\mathcal{A}$:
  $\delta := \min(\delta^*, \Delta(r))$ where r is the current round counter
  assert $(\mathsf{outp}_0, \mathsf{outp}_1)$ has been stored
  send outp_b to $\mathcal{A}$ immediately where b corresponds to the corrupt party
  delay send outp_{1-b} to the honest party in exactly δ rounds

Figure 10: The ideal fair two-party computation functionality. Depending on the protocol, the "delay send" operation may optionally require the honest party to poll before sending the output.

$\Delta(\cdot)$, such that if the adversary receives outputs by round r, then the honest parties must receive outputs by round $\Delta(\cdot)$.

Henceforth in this section, when we say *efficient* protocols, we mean the following:

- There exists a fixed polynomial $g(\cdot)$ such that if all parties behave honestly, the protocol will terminate in $g(\lambda)$ rounds and all parties output the correct outcome except with negligible probability.
- If at least one party is corrupt, we do not require that honest parties terminate in a fixed polynomial number of rounds. For example, for any fixed polynomial $g(\cdot)$, the adversary can cause a longer delay such that the adversary receives outputs in round $r := g(\lambda) + 1$ — and if the protocol Δ-realizes the intended functionality, then honest parties are then guaranteed to receive outputs by round $\Delta(r)$. In other words, in the presence of corrupt parties, the running time of the protocol may depend on the running time of the adversary (and hence not bounded by any fixed polynomial).

We now define the notion of Δ-fairness. Specifically, we define a functionality $\mathcal{F}^{f,\Delta}$ which is parametrized by a function f that it computes and the fairness parameter Δ. Below we define it for 2-party protocols, and extensions to multiple parties is in the most natural manner.

Henceforth, we say that a protocol Π realizes $\mathcal{F}^f$ with Δ-fairness if Π securely realizes $\mathcal{F}^{f,\Delta}$ by Definition 2 of Appendix A.4. We say that a protocol Π UC-realizes $\mathcal{F}^f$ with Δ-fairness if Π UC-realizes $\mathcal{F}^{f,\Delta}$ by Definition 3 of Appendix A.4. We use the notation $\mathcal{F}^f$ to denote the standard, fair multi-party computation functionality that computes the function f.

As noted earlier, all of our lower bound results are stated for the weaker notion of sequentially composable multi-party computation (Definition 2) — and this makes our lower bounds stronger. By contrast, all of our upper bound results will adopt the stronger, universally composable security.

7.4 Lower Bounds for Fair 2-Party Computation

In this section, we will present two lower bounds that show the following:

1. Fairness for general functionalities is impossible if $\mathcal{G}_{\text{att}}$ is not clock-aware, even when both parties are equipped with a secure processor, and even when the adversary is only fail-stop. Although

Cleve's lower bound proof [32] can easily be adapted to this setting, we instead prove it for a contract signing functionality, since we use this as a warmup to prove the impossibility result stated next.

2. Fairness for general functionalities is impossible if $\mathcal{G}_{\text{att}}$ is indeed clock-aware; however, only one of the two parties is equipped with a secure processor. Similarly, this lower bound holds even when the adversary is only fail-stop.

As a warmup, we first prove why fairness is impossible for general functions if our $\mathcal{G}_{\text{att}}$ functionality is not clock-aware — even when both parties have secure processors. As mentioned earlier, although Cleve's lower bound [32] can easily be adapted to this setting, we prove it instead for contract signing as a warmup exercise — since we will later modify such a proof to show the impossibility of fairness for general functions in the presence of a single secure processor.

Theorem 14 (Fairness impossiblity without trusted clock.)**.** *If $\mathcal{G}_{att}$ is not clock-aware, and assume that one-way function exists, then there exists a polynomial-time function f such that no two-party, $\mathcal{G}_{att}$-hybrid protocol can securely realize $\mathcal{F}^f$ even when both parties are in the registry of $\mathcal{G}_{att}$, and even against a fail-stop adversary.*

Proof intuition. We describe the proof intuition assuming $\mathcal{G}_{\text{att}}$ is the functionality. The same proof works for $\mathcal{G}_{\text{att}}$ too. To prove the above theorem, we consider a specific contract signing functionality, i.e., the function f_{contract} takes as input two parties' public and secret keys henceforth denoted $(\mathsf{pk}_0, \mathsf{sk}_0)$ and $(\mathsf{pk}_1, \mathsf{sk}_1)$, and outputs $\mathcal{P}_b$'s signature on the message $\vec{0}$ to party $\mathcal{P}_{1-b}$, where $b \in \{0, 1\}$. Henceforth we let $\mathcal{F}_{\text{contract}} := \mathcal{F}^{f_{\text{contract}}}$.

The proof is similar to the folklore proof that demonstrates the impossibility of contract signing in the plain setting. The idea is the following: consider a $\mathcal{G}_{\text{att}}$-hybrid protocol Π that fairly realizes $\mathcal{F}_{\text{contract}}$, and without loss of generality, assume that $\mathcal{P}_1$ sends the last message in protocol Π. We now show that if $\mathcal{P}_1$ is the corrupt party and aborts prior to sending the last message, then the ideal-world simulator Sim must send sk_1 to $\mathcal{F}_{\text{contract}}$ in within fixed polynomially many rounds during the simulation — otherwise one could leverage Sim and $\mathcal{G}_{\text{att}}$ to break the signature scheme. Since $\mathcal{F}_{\text{contract}}$ will immediately output to the honest party $\mathcal{P}_0$ in the ideal-world execution, this means that in the real-world execution (against the fail-stop adversary $\mathcal{P}_1$), $\mathcal{P}_0$ must output the correct signature within a fixed polynomial number of rounds too. We therefore conclude that Π_{-1} must fairly realize $\mathcal{F}_{\text{contract}}$ too where Π_{-1} is the same as Π but with the last message removed, and where $\mathcal{P}_0$ always assumes that the last message is dropped and directly outputs. In this way, we can one by one remove the messages in the protocol Π, until we obtain a degenerate protocol that does not send any messages. In such a protocol, the only possible interactions are between the two parties and $\mathcal{G}_{\text{att}}$. It is not hard to see that since there is no direct information flow between enclaves inside $\mathcal{G}_{\text{att}}$, the degenerate protocol cannot securely realize $\mathcal{F}_{\text{contract}}$ since otherwise we can leverage either party to build a reduction that breaks the signature scheme. We thus reach a contradition, and conclude that such a Δ-fair protocol Π cannot exist in the first place.

Proof. Consider a signature scheme with the additional following properties:

- There exists an algorithm called $\mathsf{check}(\cdot, \cdot)$ such that

$$\Pr\left[(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda) : \mathsf{check}(\mathsf{pk}, \mathsf{sk})\right] = 1$$

Contract signing function f_{contract}

On receive $((\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{pk}_1), (\mathsf{sk}_1, \mathsf{pk}'_0, \mathsf{pk}'_1))$
assert $\mathsf{pk}_0 = \mathsf{pk}'_0$, $\mathsf{pk}_1 = \mathsf{pk}'_1$; for $i \in \{0,1\}$: assert $\mathsf{check}(\mathsf{pk}_i, \mathsf{sk}_i) = 1$
output $(\mathsf{outp}_0, \mathsf{outp}_1)$ where $\mathsf{outp}_1 := \mathsf{Sign}(\mathsf{sk}_0, \vec{0})$, $\mathsf{outp}_0 := \mathsf{Sign}(\mathsf{sk}_1, \vec{0})$

- For each pk in the range of Gen, there is only one sk such that $\mathsf{check}(\mathsf{pk}, \mathsf{sk}) = 1$.
- For each sk in the range of Gen, there is only one valid signature for every message $m \in \{0,1\}^*$.

Consider a $\mathcal{G}_{\text{att}}$-hybrid protocol Π that realizes a contract signing functionality $\mathcal{F}_{\text{contract}} := \mathcal{F}^{f_{\text{contract}}}$ with fairness, against any p.p.t. fail-stop adversary.

For convenience, we only consider protocols that are *non-degenerate*, i.e., when we say that protocol Π realizes a functionality $\mathcal{F}$, we mean that Π not only satisfies the security definition as in Definition 2, moreover, if neither party aborts, both parties must output the correct answer with probability $1 - \mathsf{negl}(\lambda)$ if randomly chosen keys are used as the parties inputs. More formally, we require that there exists a negligible function $\mathsf{negl}(\lambda)$ such that for any λ,

$$\Pr\left[\begin{array}{l|l} (\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{Gen}(1^\lambda), (\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Gen}(1^\lambda), & \mathcal{P}_0 \text{ and } \mathcal{P}_1 \\ \mathsf{view} \leftarrow \text{EXEC}^{\Pi, \mathcal{P}_0, \mathcal{P}_1}(1^\lambda, (\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{pk}_1), (\mathsf{sk}_1, \mathsf{pk}_0, \mathsf{pk}_1)) & : \text{output correct} \\ & \text{sigs in } \mathsf{view} \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Suppose that $\text{EXEC}^{\Pi}(x, y, \lambda)$ completes within $R(\lambda)$ rounds with probability 1 (for any inputs). Without loss of generality, assume that the protocol always completes in exactly R rounds and that $\mathcal{P}_1$ sends the last message.

Now imagine that $\mathcal{P}_0$ executes the protocol with an adversary $\mathcal{P}_1^*$ that aborts in the last round, but otherwise follows the honest algorithm and outputs whatever the honest algorithm outputs. Notice there is a $\mathcal{P}_1^*$ that can succeed in outputting a valid σ_1 with probability $1 - \mathsf{negl}(\lambda)$.

Since Π securely realizes $\mathcal{F}_{\text{contract}}$, there exists a simulator $\mathcal{S}$ such that

$$\{\text{IDEAL}^{\mathcal{F}_{\text{contract}}, \mathcal{S}}(1^\lambda, x, y, z)\}_{x,y,z} \stackrel{c}{\equiv} \{\text{EXEC}^{\Pi, \mathcal{P}_0, \mathcal{P}_1^*}(1^\lambda, x, y, z)\}_{x,y,z}$$

For this to happen, it must be the case that there exists a negligible function $\mathsf{negl}(\lambda)$ such that for every λ, for every non-uniform polynomially bounded z,

$$\Pr\left[\begin{array}{l|l} (\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{Gen}(1^\lambda), (\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Gen}(1^\lambda), & \mathcal{S} \text{ sends } \mathsf{sk}_1 \text{ to} \\ \mathsf{view} \leftarrow \text{IDEAL}^{\mathcal{F}_{\text{contract}}, \mathcal{S}}(1^\lambda, (\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{pk}_1), (\mathsf{sk}_1, \mathsf{pk}_0, \mathsf{pk}_1), z) & : \mathcal{F}_{\text{contract}} \text{ in } \mathsf{view} \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

In other words, if randomly chosen keys are used as the parties' inputs, then $\mathcal{S}$ must send sk_1 to $\mathcal{F}_{\text{contract}}$ except with negligible probability. Since if $\mathcal{S}$ does not send sk_1 to $\mathcal{F}_{\text{contract}}$, then we can leverage the combination of $\mathcal{S}$ and $\mathcal{G}_{\text{att}}$ to construct a signature adversary that breaks the signature scheme. However, if $\mathcal{S}$ sends sk_1 to $\mathcal{F}_{\text{contract}}$, then in the ideal execution, the honest $\mathcal{P}_0$ will output the correct σ_1 with probability 1. This means that with randomly chosen keys as the parties' inputs, in the real execution $\text{EXEC}^{\Pi}(\mathcal{P}_0(1^\lambda, (\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{pk}_1)), \mathcal{P}_1^*(1^\lambda, (\mathsf{sk}_1, \mathsf{pk}_0, \mathsf{pk}_1), z))$, the honest $\mathcal{P}_0$ will output the correct σ_1 with probability $1 - \mathsf{negl}(\lambda)$. More formally, there exists a negligible function $\mathsf{negl}(\lambda)$ such that for every λ, for every non-uniform polynomially bounded z,

$$\Pr\left[\begin{array}{l}(\mathsf{pk}_0,\mathsf{sk}_0)\leftarrow\mathsf{Gen}(1^\lambda),(\mathsf{pk}_1,\mathsf{sk}_1)\leftarrow\mathsf{Gen}(1^\lambda),\\ \mathsf{view}\leftarrow\text{EXEC}^{\Pi,\mathcal{P}_0,\mathcal{P}_1^*}(1^\lambda,(\mathsf{sk}_0,\mathsf{pk}_0,\mathsf{pk}_1),(\mathsf{sk}_1,\mathsf{pk}_0,\mathsf{pk}_1),z)\end{array} : \begin{array}{l}\mathcal{P}_0 \text{ outputs}\\ \text{correct sig in}\\ \mathsf{view}\end{array}\right]\geq 1-\mathsf{negl}(\lambda)$$

This means that the last (i.e., R-th) message is superfluous. In other words, consider a protocol Π_{-1} which is identical as Π except with the last message removed. We claim that if Π realizes $\mathcal{F}_{\text{contract}}$ with fairness against any p.p.t. fail-stop adversary, then so does Π_{-1}. First, observe that Π_{-1} clearly is non-degenerate as argued above. For any (x,y,z), for any p.p.t. fail-stop adversary $\mathcal{A}$ for the protocol Π_{-1} — note that $\mathcal{A}$ can also be regarded as an adversary for the protocol Π — we have that

$$\{\text{EXEC}^{\Pi_{-1},\mathcal{A}}(\lambda,x,y,z)\}_{x,y,z}\stackrel{c}{\equiv}\{\text{EXEC}^{\Pi,\mathcal{A}}(\lambda,x,y,z)\}_{x,y,z}$$

Since Π realizes $\mathcal{F}_{\text{contract}}$ with fairness against fail-stop adversaries, there exists a p.p.t. $\mathcal{S}$, such that

$$\begin{array}{rcl}\{\text{EXEC}^{\Pi_{-1},\mathcal{A}}(\lambda,x,y,z)\}_{x,y,z}&\stackrel{c}{\equiv}&\{\text{EXEC}^{\Pi,\mathcal{A}}(\lambda,x,y,z)\}_{x,y,z}\\ &\stackrel{c}{\equiv}&\{\text{IDEAL}^{\mathcal{F}_{\text{contract}},\mathcal{S}}(\lambda,x,y,z))\}_{x,y,z}\end{array}$$

Now we can prove by induction: one at a time, we remove the rounds of the protocol Π, until we have the empty protocol $\Pi_\emptyset$ that does not send any messages; and we conclude that $\Pi_\emptyset$ securely realizes $\mathcal{F}_{\text{contract}}$ as well (and is non-degenerate) — but this clearly is impossible since otherwise we can leverage either party to break the signature scheme.

□

Theorem 15 (Fairness impossibility with trusted clocks, but when one party does not have a secure processor)**.** *Assume that one-way functions exist. Even when $\mathcal{G}_{att}$ is indeed clock-aware, there exists a polynomial-time function f such that no $\mathcal{G}_{att}$-hybrid protocol can securely realize $\mathcal{F}^f$ with Δ fairness for any polynomial function Δ, even when the adversary is only fail-stop — if only one of the parties is in the registry of $\mathcal{G}_{att}$.*

Proof intuition. Although the proof of this theorem bears a superficial resemblance to that of Theorem 14, here the proof is more subtle. Following the proof of Theorem 14, we use the same strategy of removing the protocol messages one by one starting from the last round. Assume that a $\mathcal{G}_{\text{att}}$-hybrid protocol Π securely realizes $\mathcal{F}_{\text{contract}}$ with Δ-fairness, and let $g(\cdot)$ denote the running time of Π when both parties are honest. Without loss of generality, assume that $\mathcal{P}_0$ is the party to send the last message in Π. Further, assume that $\mathcal{P}_0$ is equipped with a secure processor but $\mathcal{P}_1$ is not.

1. Now consider a fail-stop adversary $\mathcal{P}_0$ that aborts prior to sending the last message. Clearly, $\mathcal{P}_0$ can output the correct outcome in $g(\lambda)$ rounds. This means that in the simulation, by round $g(\lambda)$ the simulator Sim must send (sk_0,δ^*) to the ideal functionality $\mathcal{F}_{\text{contract}}$ (without loss of generality, we assume that the adversary $\mathcal{P}_0$ does not wait extra rounds to output after dropping the last message), since otherwise it is not difficult to construct a reduction that leverages Sim and $\mathcal{G}_{\text{att}}$ to break the signature scheme. This means that in the ideal execution, the honest party $\mathcal{P}_1$ must output to $\mathcal{Z}$ the correct output within $\Delta(g(\lambda))$ rounds — and therefore in the real execution, this must hold as well. Since $\mathcal{P}_1$ does not have a secure processor, it is not hard

to see that if $\mathcal{P}_1$ can correctly output to $\mathcal{Z}$ in $\Delta(g(\lambda))$ rounds, it can correctly output to $\mathcal{Z}$ in $g(\lambda)$ rounds — more specifically, since it did not receive any additional message after round $g(\lambda)$, it can simply complete the remaining (polynomial amount) of the computation by fast forwarding its clock.

2. Now, consider the protocol Π_{-1} that is defined in the same way as Π, but with the last message removed, and with $\mathcal{P}_1$ simply assuming that there is no last message and directly outputting without waiting to receive the last message. Using the observation from before, we know that $\mathcal{P}_1$ can output in $g(\lambda)$ rounds. Now consider a fail-stop $\mathcal{P}_1$ that aborts prior to sending the last message of Π_{-1} This means that in the simulation, the simulator Sim must send $(\mathsf{sk}_1, \delta^*)$ to $\mathcal{F}_{\text{contract}}$ by round $g(\lambda)$. And therefore, both the ideal-world and real-world $\mathcal{P}_0$ must correctly output to $\mathcal{Z}$ by round $\Delta(g(\lambda))$.

3. Now consider the protocol Π_{-2} with one additional message removed. If a fail-stop $\mathcal{P}_0$ aborts prior to sending the last message, we know that can output correctly to $\mathcal{Z}$ in $\Delta(g(\lambda))$ rounds. This means that Sim must send $(\mathsf{sk}_0, \delta^*)$ to $\mathcal{F}_{\text{contract}}$ by round $\Delta(g(\lambda))$. Therefore, in both the real- and ideal-world executions, $\mathcal{P}_1$ must output correctly to $\mathcal{Z}$ by round $\Delta(\Delta(g(\lambda)))$ However, since $\mathcal{P}_1$ does not have a secure processor, without loss of generality, $\mathcal{P}_1$ can actually output correctly to $\mathcal{Z}$ by round $g(\lambda)$.

4. Now consider the protocol Π_{-3}, and so on so forth.

Eventually we will arrive at a degenerate protocol that does not send messages between $\mathcal{P}_0$ and $\mathcal{P}_1$, and we conclude that this degenerate protocol securely realizes $\mathcal{F}_{\text{contract}}$ with Δ-fairness. The only interactions left in this degenerate protocol are between $\mathcal{G}_{\text{att}}$ and the parties. Now, since $\mathcal{G}_{\text{att}}$ ensures non-interference between enclaves, such a degenerate protocol clearly cannot securely realize $\mathcal{F}_{\text{contract}}$ since otherwise we can leverage either party to break the signature scheme. We thus reach a contradition, and conclude that such a Δ-fair protocol Π cannot exist in the first place.

Why the proof breaks when both parties have a secure processor. We point out why this proof would break when both parties are equipped with a secure processor with trusted clock — indeed, we will later show a construction that lets us achieve Δ-fairness for securely computing general functions where $\Delta(r) = 2r$.

Notice that in the above proof sketch, we repeatedly make use of the fact that when the party without a secure processor $\mathcal{P}_1$ is the honest party, when the other party aborts, in both the ideal- and real-world executions, $\mathcal{P}_1$ can output correctly within $g(\lambda)$ number of rounds — since nothing prevents $\mathcal{P}_1$ from finishing the remainder of the computation immediately and outputting. If this were not the case, e.g., if $\mathcal{P}_1$ also has a secure processor, then it would no longer be true that $\mathcal{P}_1$ can immediately output if $\mathcal{P}_0$ aborted prior to sending the last message — since $\mathcal{G}_{\text{att}}$ could now withhold messages from $\mathcal{P}_1$ for some number of rounds before outputting them to $\mathcal{P}_1$. This would cause the simulator's runtime to blow up by a polynomial factor every time we remove a message, and therefore depending on what the Δ function is, after polynomially many such removals, the simulator would no longer be polynomial time.

Proof. Imagine that f is a contract signing functionality as defined earlier. f checks that both parties provide the same public keys as input, and checks that the secret key each party provides

agrees with one of the public keys. If so, f will allow two parties to exchange signatures on the message $\vec{0}$.

Without loss of generality, assume that $\mathcal{P}_0$ has a secure processor, and $\mathcal{P}_1$ does not.

Now consider a protocol Π that securely realize $\mathcal{F}_{\text{contract}}^{\Delta}$. Consider an adversary controlling one of the parties that aborts prior to sending the last message, but otherwise obeys the honest protocol and outputs the final answer. There are two cases:

Case 1: Party who sends last message has a secure processor. Consider an adversary $\mathcal{P}_0^*$ that aborts prior to sending the last message. $\mathcal{P}_0^*$ is otherwise honest and after aborting, it follows the honest algorithm (possibly for multiple rounds, sending more inputs to $\mathcal{G}_{\text{att}}$), and finally outputs what the honest algorithm outputs (that is, the correct signature σ_1).

Since Π securely realizes $\mathcal{F}_{\text{contract}}$, there exists a simulator Sim such that

$$\{\text{IDEAL}^{\mathcal{F}_{\text{contract}},\mathsf{Sim}}(1^\lambda, x, y, z)\}_{x,y,z} \stackrel{c}{\equiv} \{\text{EXEC}^{\Pi,\mathcal{P}_0^*,\mathcal{P}_1}(1^\lambda, x, y, z)\}_{x,y,z}$$

Notice that in the real-world execution, $\mathcal{P}_0^*$ outputs the correct signature in $g(\lambda)$ number of rounds where g is a fixed polynomial. For the ideal-world execution to be computationally indistinguishable, in some round Sim must send $(\mathsf{sk}_0, \delta^*)$ to $\mathcal{F}_{\text{contract}}^{\Delta}$ for some δ^* within $g(\lambda)$ number of rounds. Since if Sim does not send sk_0 to $\mathcal{F}_{\text{contract}}$, then we can leverage the combination of Sim and $\mathcal{G}_{\text{att}}$ to construct a signature adversary that breaks the signature scheme. We know that $\delta \leq \Delta(g(\lambda))$ is bounded by a fixed polynomial. This means that in the real-world execution, $\mathcal{P}_1$ must output the correct signature within a fixed polynomial number of rounds if $\mathcal{P}_0^*$ aborted before sending the last message.

Now we consider the protocol Π_{-1} that is almost identical to Π, but with the last message removed. Further, $\mathcal{P}_1$ outputs the outcome pretending that $\mathcal{P}_0$ aborted before sending the last message in Π. We can easily show that if Π securely realizes $\mathcal{F}_{\text{contract}}^{\Delta}$ against any fail-stop adversary, so must Π_{-1}.

Case 2: Party who sends the last message does not have a secure processor. Consider an adversary $\mathcal{P}_1^*$ that aborts prior to sending the last message, but is otherwise honest — however, once it aborts, it immediately outputs what the honest algorithm would have output without waiting for more rounds. We note that since $\mathcal{P}_1^*$ does not have a secure processor, without loss of generality, it can always immediately finish all remaining computation after aborting even if the honest protocol may stipulate that the party waits for more rounds — it turns out that this is important for the induction steps in the proof to work, since otherwise the simulator's runtime will blow up by a polynomial factor with each step of the induction, and after polynomially many induction steps, the simulator's runtime may no longer be polynomial.

There exists a p.p.t. simulator Sim such that the ideal- and real-world executions are computationally indistinguishable. Further, in $g(\lambda)$ number of rounds Sim must send $(\mathsf{sk}_1, \delta^*)$ to the ideal functionality, since otherwise we can leverage Sim to break the signature scheme. We know that now the honest $\mathcal{P}_0$ will receive output by round $\Delta(g(\lambda))$. This means that in the real-world execution, $\mathcal{P}_0$ will output the correct signature by round $\Delta(g(\lambda))$ too if $\mathcal{P}_1^*$ aborted before sending its last message.

Now consider the protocol Π_{-1} which is Π but with the last message removed. Further, at the end $\mathcal{P}_0$ will compute the output as if $\mathcal{P}_1$ aborted before sending the last message in protocol Π. We can easily show that if Π securely realizes $\mathcal{F}_{\text{contract}}^{\Delta}$ against fail-stop adversaries, so must Π_{-1}.

Induction. Based on the above, we can do an induction, removing the protocol messages one by one from the last message till the first. In each step of the induction, we have a protocol with the guarantee that if both parties are honest, both parties will output the correct signature within a fixed polynomial number of rounds — note that whenever the reduction comes to a step when the corrupt party does not have a secure processor, without loss of generality, the corrupt party can always finish all remainding computation and output immediately. By leveraging this property, in any step of the induction, the protocol's running time is bounded by $\Delta(g(\lambda))$ when both parties are honest. At the end of the induction, we conclude that one party can output the correct signature in a fixed polynomial number of rounds, without receiving any messages from the other party. This clearly is impossible with $\mathcal{G}_{\text{att}}$ or $\mathcal{G}_{\text{att}}$, since no information flow exists between the two parties' enclaves within $\mathcal{G}_{\text{att}}$ or $\mathcal{G}_{\text{att}}$ itself. □

7.5 Fair 2-Party Coin Toss with a Single Secure Processor

Although we cannot Δ-fairly compute general functionalities when only one party has an attested execution processor, we show that interestingly, even with only one attested execution processor (that has a trusted clock), we can already Δ-fairly compute more functions than what is known to be possible in the plain setting. Specifically, we show how to realize a Δ-fair 2-party coin toss protocol. Due to a well-known result by Cleve [32], we know that fair 2-party coin toss is impossible in the standard setting even against a fail-stop adversary.

A two-party coin toss function is defined as below, and henceforth we define the functionality $\mathcal{F}_{\text{coin}} := \mathcal{F}^{f_{\text{coin}}}$.

Coin toss function f_{coin}

On receive $(\mathsf{inp}_0, \mathsf{inp}_1)$:
if $\mathsf{inp}_0 = \mathsf{inp}_1 =$ "okay": let $\mathsf{coin} \xleftarrow{\$} \{0,1\}$, $\mathsf{outp}_0 = \mathsf{outp}_1 := \mathsf{coin}$
else if one input $\neq$ "okay": for $b \in \{0,1\}$, let $\mathsf{outp}_b \xleftarrow{\$} \{0,1\}$
output $(\mathsf{outp}_0, \mathsf{outp}_1)$

More specifically, the fair coin toss functionality $\mathcal{F}_{\text{coin}}$ tosses a fair coin between two parties, such that 1) if both parties are honest, they receive the same uniform random coin as output; and 2) if one of the parties is corrupt and deviates from the protocol (including aborting), the other honest party outputs a fresh, independent random coin.

We now describe a $\mathcal{G}_{\text{att}}$-hybrid protocol that UC-realizes the $\mathcal{F}_{\text{coin}}$ functionality with Δ fairness where where $\Delta(r) := r + 1$. The formal description of the protocol is presented in Figure 11. Here we explain the intuition. Imagine that a server $\mathcal{S}$ has a secure processor (modeled as $\mathcal{G}_{\text{att}}$) but the client $\mathcal{C}$ does not. First, the client $\mathcal{C}$ establishes a secure channel with an $\mathcal{G}_{\text{att}}$ enclave. Then, $\mathcal{G}_{\text{att}}$ flips a random coin and sends it over the secure channel to $\mathcal{C}$ — note that $\mathcal{S}$ is the intermediary forwarding the message, but $\mathcal{S}$ cannot see the contents of this encrypted message nor modify it. At this moment, however, $\mathcal{G}_{\text{att}}$ still withholds the coin from $\mathcal{S}$, such that $\mathcal{S}$ cannot decide whether to drop this message based on the value of the coin — only in the next round will $\mathcal{G}_{\text{att}}$ reveal the outcome of the coin to $\mathcal{S}$. Now if $\mathcal{S}$ fails to forward the message in time, $\mathcal{C}$ simply treats $\mathcal{S}$ as having aborted, flips an independent random coin on its own, and outputs its value.

In comparison, in standard coin flipping without $\mathcal{G}_{\text{att}}$, the party who sees the outcome of the

$\mathsf{prog}_{\text{coin}}[\mathcal{C}, \mathcal{S}]$

On input $(\text{“toss”}, g^a)$:
  let $b \xleftarrow{\$} \mathbb{Z}_p$, $\mathsf{sk} := (g^a)^b$, $\mathsf{coin} \xleftarrow{\$} \{0,1\}$, $\mathsf{ct} := \mathsf{Enc}_{\mathsf{sk}}(\mathsf{coin})$, return (ct, g^a, g^b)

On input $(\text{“output”}, v)$:
  if $v \neq \bot$, return v
  assert at least 1 round has been skipped since “toss”, return coin

$\mathbf{Prot}_{\text{coin}}[sid, \mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{C}, \mathcal{S}]$

Server $\mathcal{S}$:

  let $eid := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{coin}}[\mathcal{C}, \mathcal{S}])$, await $(\text{“toss”}, g^a)$ from $\mathcal{C}$
  $(\mathsf{ct}, g^a, g^b, \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“toss”}, g^a))$, send $(eid, \psi(\mathcal{C}, \mathsf{ct}, g^a, g^b, \sigma))$ to $\mathcal{C}$
  skip a round
  $(\mathsf{coin}, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”})$
  except: if $\mathcal{C}$ aborted, $\mathsf{coin} \xleftarrow{\$} \{0,1\}$
  output coin

Client $\mathcal{C}$:

  let $a \xleftarrow{\$} \mathbb{Z}_p$, send $(\text{“toss”}, g^a)$ to $\mathcal{S}$, await (eid, ψ) from $\mathcal{S}$
  parse $\psi := (\mathsf{msg}, C, \pi)$, assert $\mathsf{NIZK.Ver}(\mathsf{crs}, (sid, eid, C, \mathsf{mpk}, \mathcal{G}_{\text{acrs}}.\mathsf{mpk}, \mathcal{C}, \mathsf{msg}), \pi)$
  parse $\mathsf{msg} := (\mathsf{ct}, g^a, \_)$, if parse succeeds: let $\mathsf{coin} := \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct})$
  except: if $\mathcal{S}$ aborted or no coin was decrypted: $\mathsf{coin} \xleftarrow{\$} \{0,1\}$
  output coin

Figure 11: Fair 2-party coin toss when only $\mathcal{S}$ has a secure processor. ψ produces a witness indistinguishable proof that either a ciphertext encrypts a valid attestation for the message, or it encrypts the receiver's identity key — see Section 6 for detailed definitions. Assertion failures are caught by the exception handler **except**. *The await instruction waits to receive the message at the beginning of the next round, and treats the other party as having aborted if such a message was not received in time. If the other party aborted during the protocol, control is immediately passed to line marked* **except**.

coin flip first can decide whether to abort based on whether he likes the outcome. Even though the other party can generate a fresh random coin at this moment, the outcome will already be biased. Our protocol circumvents this problem because a corrupt server $\mathcal{S}$ effectively must decide whether to abort before seeing the outcome of the coin toss.

Theorem 16 (Fair 2-party coin toss with a single clock-aware attested execution processor)**.** *Assume that the encryption scheme is a perfectly correct and satisfies semantic security, assume that the DDH assumption holds in the relevant group, and that the signature scheme Σ is existentially unforgeable under chosen message attacks, then the $\mathcal{G}_{att}$-hybrid protocol defined in Figure 11 UC-realizes $\mathcal{F}_{coin}$ with Δ fairness where $\Delta(r) := r + 1$.*

Proof. We now prove the above theorem.

When $\mathcal{S}$ is corrupt. We can construct the following simulator Sim.

- Sim randomly generates a and sends $\mathcal{S}$ the tuple $(\text{“toss”}, g^a)$.
- Sim directly passes through any communication between $\mathcal{S}$ and $\mathcal{G}_{\text{att}}$, except if the call is $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”})$ where eid is the challenge eid — this case will be treated later.
- If $\mathcal{S}$ sends the tuple (eid, ψ) in time where $\psi := (\mathsf{ct}, g^a, g^b, \_, \pi)$ — at this time eid is called the challenge eid — Sim checks to see if $\mathcal{S}$ ever made an $\mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{coin}}[\mathcal{C}, \mathcal{S}])$ query that returned the challenge eid. Sim also checks to see if $\mathcal{S}$ ever made an $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{““toss””}, g^a))$ query that returned $(\mathsf{ct}, (g^a, g^b), \_)$ where the terms $eid, \mathsf{ct}, g^a, g^b$ correspond to either the term contained in the message from $\mathcal{S}$ or what Sim sent earlier. If either check fails, Sim sends $(\text{“compute”}, \bot)$ to $\mathcal{F}_{\text{coin}}$, and then it sends $(\text{“output”}, \mathsf{now})$ where now is the identity function that indicates to output to the honest party in this very round.

  Otherwise, if the checks pass, Sim sends $(\text{“compute”}, \text{“okay”})$ to $\mathcal{F}_{\text{coin}}$, followed by $(\text{“output”}, \mathsf{now})$. At this moment Sim receives an output coin from $\mathcal{F}_{\text{coin}}$ and remembers it.
- If Sim did not receive such a tuple from $\mathcal{S}$ in time, $\mathcal{S}$ is treated as having aborted. In this case, Sim also sends $(\text{“compute”}, \bot)$ to $\mathcal{F}_{\text{coin}}$, and then it sends $(\text{“output”}, \mathsf{now})$ to $\mathcal{F}_{\text{coin}}$ where now is the identity function.
- If $\mathcal{S}$ queries $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, v)$, if $v \neq \bot$, simply pass through the call. If $v = \bot$ and Sim has received coin from $\mathcal{F}_{\text{coin}}$, let $(\mathsf{coin}, \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, \mathsf{coin})$, return (coin, σ). Otherwise, return $\bot$.
- Finally, Sim passes through any communication between $\mathcal{Z}$ and $\mathcal{S}$.

Hybrid 0. Hybrid 0 is the same as the simulation, except that whenever $\mathcal{S}$ sends any query of the form $\mathcal{G}_{\text{att}}.\texttt{resume}(\_, (\text{“toss”}, g^a))$ pertaining to the challenge g^a that Sim sent to $\mathcal{S}$, $\mathcal{G}_{\text{att}}$ now chooses sk at random rather than by computing $(g^a)^b$. Further, if $\mathcal{S}$ queries $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“toss”}, g^a))$ on the challenge eid, then the simulated honest client will also use the same random sk $\mathcal{G}_{\text{att}}$ generates to decrypt.

Claim 15. *Assume that the DDH assumption holds, then Hybrid 0 is computationally indistinguishable from the simulated execution.*

Proof. Through a straightforward reduction to DDH — a hybrid argument can be applied for over each query of the form $\mathcal{G}_{\text{att}}.\texttt{resume}(\_, (\text{“toss”}, g^a))$. □

Hybrid 1. Hybrid 1 is almost the same as Hybrid 0, except that when Sim sends $(\text{“compute”}, \perp)$ to $\mathcal{F}_{\text{coin}}$ and the real-world client did not trigger the exception handler, we modify the simulation to simply abort.

Claim 16. *Assume that the with proof system is computationally sound, that* PKE *is perfectly correct, and that signature scheme is secure, then the probability that Hybrid 1 aborts is negligible.*

Proof. Similar to that of Lemma 1 of Section 6. □

We now consider two cases, and in both cases, we show why Hybrid 1 is computationally indistinguishable from the real execution.

- Case (a): the random coin $\mathcal{F}_{\text{coin}}$ generated for the corrupt party is the same as coin $\mathcal{G}_{\text{att}}$ generated; and
- Case (b): the random coin $\mathcal{F}_{\text{coin}}$ generated for the corrupt party differs from the coin $\mathcal{G}_{\text{att}}$ generated

Hybrid 2(a). Almost the same as Hybrid 1, except that we switch the random keys sks back to real keys again.

Claim 17. *Given that the DDH assumption holds, Hybrid 2(a) is computationally indistinguishable from Hybrid 1.*

Proof. Similar to the indistinguishability of Hybrid 0 and the simulated execution. □

Claim 18. *Conditioned on Case (a) happening and that the simulation did not abort, it holds that Hybrid 2(a) is identically distributed as the real execution.*

Proof. Straightforward to observe. □

Henceforth, we focus on Case (b).

Hybrid 2(b). Hybrid 2(b) is almost identical as Hybrid 1, except that $\mathcal{G}_{\text{att}}$ now encrypts $1 - \mathsf{coin}$ instead of coin.

Claim 19. *Assume that the encryption scheme is semantically secure, then Hybrid 2(b) is computationally indistinguishable from Hybrid 1.*

Proof. Through a straightforward reduction to encryption security. □

Hybrid 3(b). Hybrid 3(b) is almost identical as Hybrid 2(b), except that now we switch back to real sks rather than random sks.

Claim 20. *Assume that the DDH assumption holds, then Hybrid 3(b) is computationally indistinguishable from Hybrid 2(b).*

Proof. Symmetric to the indistinguishability of Hybrid 2(a) and Hybrid 1. □

Claim 21. *Conditioned on Case (b) happening and that the simulation did not abort, Hybrid 3(b) is identically distributed from the real execution.*

Proof. Straightforward to verify. □

When $\mathcal{C}$ is corrupt. We can construct the following simulator Sim. Sim receives a tuple ("toss", g^a) from $\mathcal{C}$. If $\mathcal{C}$ aborted without sending a well-formed message, Sim sends ("compute", $\perp$) to $\mathcal{F}_{\text{coin}}$ followed by ("output", now).

Otherwise, Sim sends ("compute", "okay") to $\mathcal{F}_{\text{coin}}$ followed by ("output", $\mathsf{now}+1$), and receives an outcome coin from $\mathcal{F}_{\text{coin}}$. Now Sim uses the witness indistinguishable proof, but uses $\mathcal{C}$'s identity key as the alternative witness, to compose an appropriate answer back to $\mathcal{C}$, and it will embed the coin it received from $\mathcal{F}_{\text{coin}}$ into this answer.

Sim passes through any communication between $\mathcal{C}$ and $\mathcal{Z}$.

It is not hard to see that given that the proof system is witness indistinguishable and that PKE is perfectly correct, the simulated execution and the real-world execution are computationally indistinguishable. The formal proof is straightforward and is similar to the honest-server case of our $(\mathcal{G}_{\text{acrs}}, \mathcal{G}_{\text{att}})$-hybrid protocol that achieves MPC with a single secure processor (see Section 6).

When both parties are honest. It is not hard to see that this case can be simulated easily given our usage of secure and authenticated channels in the protocol.

□

7.6 Fair Generic 2-Party Computation When Both Have Secure Processors

We now show that if both parties have a clock-aware attested execution processor, we can securely compute any polynomial-time function with Δ fairness. The formal protocol description is presented in Figure 12. We explain the intuition below. The idea is to have a *quid pro quo* style protocol, where the two parties' secure processors bargain with each other as to when to release the computation result. The protocol works as follows:

- In the beginning of the protocol, the two parties' secure processors establish secure channels with each other, such that both secure processors learn the outcome. At this time, both secure processors set $\delta := 2^\lambda$, which is the intended time to release the result to the platform that owns the secure processor.
- Now, in every turn, each secure processor halves δ and promises the other secure processor that it will release the outcome in $\delta \leftarrow \frac{\delta}{2}$ time instead.
- If both parties are honest, then after $O(\lambda)$ rounds, both parties learn the outcome. If one of the parties (say, $\mathcal{P}_0$) aborts, then it is not hard to see that if $\mathcal{P}_0$ learns the outcome by round r, then $\mathcal{P}_1$ learns the outcome by round $2r$.

$\mathsf{prog}_{\text{fair2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, i]$ where $i \in \{0,1\}$ // for party $\mathcal{P}_i$

On initialize: $\delta := 2^\lambda$

On input ("keyex"): let $a \xleftarrow{\$} \mathbb{Z}_p$, and return g^a

On input ("send", g^b, inp_i):
  assert "keyex" has been called
  let $\mathsf{sk} := (g^b)^a$, $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(\mathsf{inp}_i)$, return ct

On input ("receive", ct):
  assert "keyex" and "send" have been called, assert ct not seen
  let $x_{1-i} := \mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct})$, $\mathsf{ct}' := \mathsf{AE.Enc}_{\mathsf{sk}}(\delta)$, return ct'

On input* ("ack", ct):
  assert ct not seen, assert $\mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct}) = \delta$
  let $\delta := \lfloor \delta/2 \rfloor$, $\mathsf{ct}' := \mathsf{AE.Enc}_{\mathsf{sk}}(\delta)$, return ct'

On input ("output", v):
  if $v \neq \perp$, return v
  assert "receive" has been called, assert current round $\geq \delta$, return $f(\mathsf{inp}_0, \mathsf{inp}_1)$

$\mathbf{Prot}_{\text{fair2pc}}[sid, f, \mathcal{P}_0, \mathcal{P}_1, i]$ where $i \in \{0,1\}$ //for party $\mathcal{P}_i$

On input inp_i from $\mathcal{Z}$:
  let $eid := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{fair2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, i])$
  let $(g^a, \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{"keyex"})$
  send (eid, g^a, σ) to $\mathcal{P}_{1-i}$, await (eid', g^b, σ') from $\mathcal{P}_{1-i}$
  assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((sid, eid', \mathsf{prog}_{\text{fair2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, 1-i], g^b), \sigma')$
  let $(\mathsf{ct}, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{"send"}, g^b, \mathsf{inp}_i))$, send ct to $\mathcal{P}_{1-i}$, await ct'
  let $(\mathsf{ct}, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{"receive"}, \mathsf{ct}'))$, send ct to $\mathcal{P}_{1-i}$, await ct'
  repeat λ times: let $(\mathsf{ct}, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{"ack"}, \mathsf{ct}'))$, send ct to $\mathcal{P}_{1-i}$, await ct'

On input "output" from $\mathcal{Z}$:
  return $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{"output"}, \perp)$

Figure 12: Fair 2-party computation when both parties have secure processors. Upon assertion failure, control is yielded back to the environment $\mathcal{Z}$. *Await instructions yield control back to the environment $\mathcal{Z}$ if no message is received at the end of a round.*

Figure 12 describes the full protocol. Most of the protocol is quite natural, but there are a couple technicalities to point out. First, because each party sends their respective input to its local enclave, the simulator can observe this communication and perform extraction. Second, we note that the enclave program's "output" entry point offers a backdoor v for programming the output. In this protocol, however, since both parties simply query their local enclave for the output, the backdoor for programming is simpler than the non-fair counterpart in Section 5.2. Particularly, here a (corrupt) party can always choose not to learn its own output and ask the enclave to instead sign any value v of its choice, and this does not harm the security of the honest party in the protocol. By contrast, in the non-fair counterpart in Section 5.2, one party obtains the signed attestation and sends it to the other (in the efficient variant where only one party's enclave performs program-dependent computation). Therefore, in that protocol the honest party must protect itself against a corrupt party who might aribtrarily program its enclave's output.

Theorem 17 (Fair 2-party computation when both have clock-aware attested execution processors)**.** *Assume that DDH holds in the relevant group,* AE *is perfectly correct, and satisfies semantic security and INT-CTXT security, and that the signature scheme* Σ *is existentially unforgeable under chosen message attacks, it holds that the protocol described in Figure 12 UC-realizes* $\mathcal{F}^f$ *with* Δ *fairness, where* $\Delta(r) := 2r$*.*

Proof. We now prove the above theorem. When both parties are honest, it is not difficult to construct a simulation. We focus on the more interesting case when one party is corrupt. Without loss of generality, we consider the case when $\mathcal{P}_0$ is honest and $\mathcal{P}_1$ is corrupt — the other case is symmetric.

We can now construct a simulator Sim as described below.

- Unless otherwise noted later, Sim passes through messages between $\mathcal{P}_1$ and $\mathcal{G}_{\text{att}}$. Sim also passes through communications between $\mathcal{P}_1$ and $\mathcal{Z}$.
- Sim calls $eid' := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{fair2pc}}[f, \mathcal{P}_0, \mathcal{P}_1, 0])$, and $(g^b, \sigma') := \mathcal{G}_{\text{att}}.\texttt{resume}(eid', \text{"keyex"})$ and sends (eid', g^b, σ') to $\mathcal{P}_1$.

  Sim waits to receive the first message (eid, g^a, σ) from $\mathcal{P}_1$ — if this tuple was not the answer to a previous $\mathcal{G}_{\text{att}}$ query, jump to the exception handler denoted **except**. At this point, eid is called the challenge eid.
- The first time $\mathcal{P}_1$ calls $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{"send"}, g^b, \mathsf{inp}_1))$ for some input inp_1 where eid is the challenge eid, and g^b is what Sim has sent, the simulator Sim extracts and sends inp_1 to $\mathcal{F}^{f,\Delta}$. Note that this message may be received before Sim receives (eid, g^a, σ) from $\mathcal{P}_1$. If this is the case, the extraction actually happens when Sim receives (eid, g^a, σ) from $\mathcal{P}_1$ — this can be achieved if Sim simply remembers every $\mathcal{G}_{\text{att}}$ call $\mathcal{P}_1$ has made.
- Sim calls $(\mathsf{ct}_1, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid', (\text{"send"}, g^a, \vec{0}))$ and sends ct_1 to $\mathcal{P}_1$.

  Sim waits to receive ct from $\mathcal{P}_1$. If $(\mathsf{ct}, \_)$ is the not the result of the first $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{"send"}, g^b, \_))$ call where eid is the challenge eid, and g^b was what Sim previously sent to $\mathcal{P}_1$, or if no such call has taken place, then go to the exception handler **except**.
- Sim calls $(\mathsf{ct}_2, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid', (\text{"receive"}, \mathsf{ct}))$ and sends ct_2 to $\mathcal{P}_1$.

Sim waits to receive ct from $\mathcal{P}_1$. If $(\mathsf{ct}, \_)$ is the not result of the first $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“receive”}, g^b, \mathsf{ct}_1))$ call where eid is the challenge eid, and g^b, ct_1 are what Sim previously sent to $\mathcal{P}_1$, or if no such call has taken place, then go to the exception handler **except**.

- Now repeat λ times with i starting at 3 and incrementing each turn: Sim calls $(\mathsf{ct}_i, \_) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid', (\text{“ack”}, \mathsf{ct}))$, and sends ct_i to $\mathcal{P}_1$. Sim waits to receive ct from $\mathcal{P}_1$. If ct is the not result of the first $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“ack”}, \mathsf{ct}_{i-1})$ call where eid is the challenge eid, and ct_{i-1} is what Sim just sent $\mathcal{P}_1$, then go to the exception handler **except**.
- Sim simulates the current value of δ: every time it receives an acknowledgement from $\mathcal{P}_1$, and the exception handler is not triggered, it halves δ.
- If $\mathcal{P}_1$ ever sends any message late, it is treated as having aborted. In this case, jump to the exception handler denoted **except**.
- If the execution completes without triggering the exception handler, then send $(\text{“output”}, \mathsf{now})$ to $\mathcal{F}^{f,\Delta}$. At this time Sim receives the outcome of the computation from $\mathcal{F}^{f,\Delta}$ henceforth denoted outp_1.
- **except**: If any exception was triggered: in round $\frac{\delta}{2}$, send $(\text{“output”}, \Delta)$ where $\Delta(r) := 2r$ to $\mathcal{F}^{f,\Delta}$. At this time Sim receives the outcome of the computation from $\mathcal{F}^{f,\Delta}$ henceforth denoted outp_1.
- When Sim receives $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, \text{“output”}, v)$ from $\mathcal{P}_1$, if $v \neq \bot$, pass through the call. Else if current round $< \frac{\delta}{2}$, return $\bot$. Else, call $(\mathsf{outp}_1, \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid', \text{“output”}, \mathsf{outp}_1)$ return $(\mathsf{outp}_1, \sigma)$.

Hybrid 0. Identical to the simulation, except that every occurrence of the challenge $\mathsf{sk} = g^{ab}$ is replaced with a random key.

Claim 22. *Assume that DDH holds, then Hybrid 0 is computationally indistinguishable from the simulation.*

Proof. Straightforward reduction to DDH security. □

Hybrid 1. Identical to Hybrid 0, except that every time the exception handler is triggered in the simulation, if the real-world $\mathcal{P}_0$ would not have had an assertion failure or awaited a message that did not arrive at the end of a round, abort the simulation.

Claim 23. *Assume that the signature scheme Σ is secure and that AE has INT-CTXT security, it holds that Hybrid 1 aborts with negligible probability.*

Proof. If the exception handler is triggered in the simulation, and the real-world $\mathcal{P}_0$ did not have a signature verification failure or a ct-related failure (that is, either ct was seen before or decryption of ct did not succeed or yield the expected result), then one can easily leverage $\mathcal{P}_1$ to build a reduction that either breaks the signature scheme or the INT-CTXT security of the AE scheme. □

Hybrid 2. Identical to Hybrid 1, except that encryption of the $\vec{0}$ vector is replaced with encryption of the honest client's true input.

Claim 24. *Assume that* AE *is semantically secure, Hybrid 2 is computationally indistinguishable from Hybrid 1.*

Proof. Straightforward reduction to the semantic security of AE. □

Hybrid 3. Identical to Hybrid 2, except that the challenge sk is now replaced with the true g^{ab} again.

Claim 25. *Assume that DDH is hard, Hybrid 3 is computationally indistinguishable from Hybrid 2.*

Proof. By straightforward reduction to DDH security. □

Claim 26. *Conditioned on simulation not aborting, Hybrid 3 is identically distributed as the real execution.*

Proof. Straightforward to observe. □

□

8 Variant Models and Additional Results

8.1 Side-Channel Attacks and Transparent Enclaves

Many secure processors are known to be vulnerable to certain side-channel attacks such as cache-timing or differential power analysis. Complete defense against such side channels remains an area of active research [39–42, 58, 81].

Recently, Tramèr et al. [79] ask the question, what kind of interesting applications can we realize assuming that such side-channels are unavoidable in secure processors? Tramèr et al. [79] then propose a new model which they call the *transparent enclave* model. The transparent enclave model is almost the same as our $\mathcal{G}_{\text{att}}$, except that the enclave program leaks all internal states to the adversary $\mathcal{A}$. Nonetheless, $\mathcal{G}_{\text{att}}$ still keeps its master signing key msk secret. In practice, this model requires us to only spend effort to protect the secure processor's attestation algorithm from side channels, and we consider the entire user-defined enclave program to be transparent to the adversary.

Tramèr et al. [79] then show how to realize interesting security tasks such as cryptographic commitments and zero-knowledge proofs with only transparent enclaves, albeit in a model where the secure processor setup is assumed to be local to each protocol. In other words, Tramèr et al. prove security of their protocols only when each protocol instance freshly rekeys the secure processor's master key pair $(\mathsf{mpk}, \mathsf{msk})$. In practice, however, the secure processor's master key pair is globally shared across multiple users and applications — in fact, such reusability is an important desideratum as we point out in Section 1.

We will now show how to design a class of useful protocols when the secure processor's enclave is assumed to be transparent to the adversary. Further, we will prove security under a model when the secure processor is shared across users and applications (i.e., modeled as a global functionality). Before doing so, we first clarify the formal abstraction for transparent enclaves.

The transparent enclave functionality $\widehat{\mathcal{G}}_{\text{att}}$. We define the transparent enclave functionality $\widehat{\mathcal{G}}_{\text{att}}$ to be almost identical to $\mathcal{G}_{\text{att}}$, except that besides outputting the pair (outp, σ) to the caller upon the `resume` entry point, it also leaks to the caller all random bits internally generated during the computation.

We also point out that if an enclave program is deterministic, it is naturally transparent (even for $\mathcal{G}_{\text{att}}$), since the only source of secrets was randomness internally generated inside an enclave program (recall that this was needed for an enclave to perform key exchange with a remote client and establish a secure channel).

8.2 Composable Commitments with Transparent Enclaves

We show how to realize UC-secure commitments when both parties have a secure processor since otherwise the task would have been impossible as noted earlier. Although intuition is quite simple — the committer could commit the value to its local enclave, and later ask the enclave to sign the opening — it turns out that this natural protocol candidate is not known to have provable security. Our actual protocol involves non-trivial techniques to achieve equivocation when the receiver is corrupt, a technical issue that arises commonly in UC proofs.

Challenge in achieving equivocation. We note that because the committer must commit its value b to its local enclave, extraction is trivial when the committer is corrupt. The challenge is how to equivocate when the receiver is corrupt. In this case, the simulator must first simulate for the corrupt receiver a commitment-phase message which contains a valid attestation. To do so, the simulator needs to ask its enclave to sign a dummy value — note that at this moment, the simulator does not know the committed value yet. Later, during the opening phase, the simulator learns the opening from the commitment ideal functionality $\mathcal{F}_{\text{com}}$. At this moment, the simulator must simulate a valid opening-phase message. The simulator cannot achieve this through the normal execution path of the enclave program, and therefore we must provide a special backdoor for the simulator to program the enclave's attestation on the opened value. Furthermore, it is important that a real-world committer who is potentially corrupt cannot make use of this backdoor to equivocate on the opening.

Our idea is therefore the following: the committer's enclave program must accept a special value c for which the receiver knows a trapdoor x such that $\mathsf{owf}(x) = c$, where owf denotes a one-way function. Further, the committer's enclave must produce an attestation on the value c such that the receiver can be sure that the correct c has been accepted by the committer's enclave. Now, if the committer produces the correct trapdoor x, then the committer's enclave will allow it to equivocate on the opening. Note that in the real-world execution, the honest receiver should never disclose x, and therefore this backdoor does not harm the security for an honest receiver. However, in the simulation when the receiver is corrupt, the simulator can capture the receiver's communication with $\widehat{\mathcal{G}}_{\text{att}}$ and extract the trapdoor x. Thus the simulator is now able to program the enclave's opening after it learns the opening from the $\mathcal{F}_{\text{com}}$ ideal functionality.

The protocol works as follows:

- First, the receiver selects a random trapdoor x, and sends it to its local enclave. The local enclave computes $c := \mathsf{owf}(x)$ where owf denotes a one-way function, and returns (c, σ) where σ is an attestation for c.

$\mathsf{prog}_{\mathsf{owf}}$

On input ("trapdoor", x):
  return $\mathsf{owf}(x)$

$\mathsf{prog}_{\mathsf{com}}$

On input ("commit", b, c): store (b, c), return c

On input ("open", x, b'):
  assert a tuple (b, c) has been stored
  if $\mathsf{owf}(x) = c$ return b' else return b

$\mathbf{Prot}_{\mathsf{com}}[sid, \mathcal{R}, \mathcal{C}]$

Receiver $\mathcal{R}$:

On input "commit" from $\mathcal{Z}$:
  let $eid_R := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathsf{owf}})$
  let $x \xleftarrow{\$} \{0,1\}^\lambda$, let $(c, \sigma) := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_R, (\text{"trapdoor"}, x))$
  send (eid_R, c, σ) to $\mathcal{C}$ and wait to receive (eid_C, σ') from $\mathcal{C}$
  assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((sid, eid_C, \mathsf{prog}_{\mathsf{com}}, c), \sigma')$ and store eid_C and output "receipt"

On input "open" from $\mathcal{Z}$:
  assert "commit" has been called and wait to receive (b, σ) from $\mathcal{C}$
  assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((sid, eid_C, \mathsf{prog}_{\mathsf{com}}, b), \sigma)$ and output b

Committer $\mathcal{C}$:

On input ("commit", b) from $\mathcal{Z}$:
  wait to receive (eid_R, c, σ) from $\mathcal{R}$ and assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((sid, eid_R, \mathsf{prog}_{\mathsf{owf}}, c), \sigma)$
  let $eid_C := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathsf{com}})$
  let $(c, \sigma) := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_C, (\text{"commit"}, b, c))$
  send (eid_C, σ) to $\mathcal{R}$, output "okay"

On input "open" from $\mathcal{Z}$:
  assert "commit" has been called, let $(b, \sigma) := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_C, (\text{"open"}, \bot, \bot))$
  send (b, σ) to $\mathcal{R}$, output "okay"

Figure 13: Composable commitment: both committer and receiver have secure processors.

- Next, the committer receives (c, σ) from the receiver. If the attestation verifies, it then sends to its enclave the bit b to be committed, along with the value c that is the outcome of the one-way function over the receiver's trapdoor x. The committer's secure processor now signs the c value received in acknowledgment, and the receiver must check this attestation to make sure that the committer did send the correct c to its own enclave.

- Next, during the opening phase, the committer can ask its local enclave to sign the opening of the committed value, and demonstrate the attestation to the receiver to convince him of the opening. Due to a technicality commonly referred to as "equivocation" that arises in UC proofs, the enclave's "open" entry point provides the following backdoor: if the caller provides a pair of values (x, b') such that $\mathsf{owf}(x) = c$ where c was stored earlier by the enclave, then the enclave will sign b' instead of the previously committed value b.

Theorem 18 (Composable commitment with transparent enclaves)**.** *Assume that* owf *is a secure one-way function, and* Σ *is secure, the protocol described in Figure 13 UC-realizes* $\mathcal{F}_{com}[sid, \mathcal{C}, \mathcal{R}]$ *where* $\mathcal{R}$ *denotes the receiver, and* $\mathcal{C}$ *denotes the committer.*

Proof. We consider three cases, where either the receiver $\mathcal{R}$, the committer $\mathcal{C}$, or both are honest. In all cases, we consider an ideal-world adversary (the simulator Sim) that internally runs a copy of a real-world adversary $\mathcal{A}$ and emulates an interaction between $\mathcal{A}$ and honest parties running $\mathbf{Prot}_{\mathsf{com}}$.

Sim forwards any messages sent between $\mathcal{Z}$ and $\mathcal{A}$. When $\mathcal{A}$ wants to call $\widehat{\mathcal{G}}_{\text{att}}$, Sim records the message, forwards it to $\widehat{\mathcal{G}}_{\text{att}}$, and records the response.

Committer is corrupt: We first describe the ideal-world simulator Sim. Sim runs a copy of the real-world adversary $\mathcal{A}$ and simulates an execution of $\mathbf{Prot}_{\mathsf{com}}$ with an honest $\mathcal{R}$. In particular, Sim emulates $\mathcal{R}$'s calls to $\widehat{\mathcal{G}}_{\text{att}}$. Observe that the responses sent by $\widehat{\mathcal{G}}_{\text{att}}$ are anonymous, and since both the committer and receiver are in the registry (i.e., have secure processors), the responses are identically distributed no matter who the caller is.

Specifically, Sim installs an enclave with id eid_R running $\mathsf{prog}_{\text{owf}}$ in $\widehat{\mathcal{G}}_{\text{att}}$, obtains (c, σ) by resuming the enclave with a random input x, and sends (eid_R, c, σ) to the corrupted committer. Then, when $\mathcal{A}$ sends (eid_C, σ) to $\mathcal{R}$, let c be the challenge that Sim sent to $\mathcal{A}$ on behalf of $\mathcal{R}$. Then, the simulator aborts and outputs sig-failure1 in the following cases:

- $\mathcal{A}$ never invoked $\widehat{\mathcal{G}}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\mathsf{com}})$ with eid_C as response; or
- (c, σ) was never output by $\widehat{\mathcal{G}}_{\text{att}}$ on a valid call $\widehat{\mathcal{G}}_{\text{att}}.\texttt{resume}(eid_C, (\text{“commit”}, b, c))$ from $\mathcal{A}$.

If Sim does not abort, it looks up this value b and sends $(\text{“commit”}, b)$ to $\mathcal{F}_{\text{com}}$ on $\mathcal{C}$'s behalf. Note that such a b must exist (as otherwise Sim aborts) and it must be unique (as $\mathsf{prog}_{\mathsf{com}}$ has non-reentrant entry points, i.e., it outputs $\perp$ if queried more than once). Finally, when $\mathcal{A}$ sends (b', σ) to $\mathcal{R}$, Sim aborts and outputs sig-failure2, if (b', σ) was not a response of $\widehat{\mathcal{G}}_{\text{att}}$ on a call by $\mathcal{A}$ of the form $\widehat{\mathcal{G}}_{\text{att}}.\texttt{resume}(eid_C, (\text{“open”}, \_, \_))$. Otherwise, if $b' \neq b$, where b is the value that Sim sent to $\mathcal{F}_{\text{com}}$, Sim aborts and outputs owf-failure. In all other cases, Sim continues the simulation and sends “open” to $\mathcal{F}_{\text{com}}$ on $\mathcal{C}$'s behalf.

Indistinguishability: As the simulator emulates the honest receiver $\mathcal{R}$ perfectly for $\mathcal{A}$, conditioned on Sim not aborting, the views of $\mathcal{A}$ in a real execution and in the ideal-world execution above are identically distributed. Furthermore, conditioned on Sim not aborting in the simulation, it is immediate that $\mathcal{A}$ eventually opens the value b that Sim “extracted” and sent to $\mathcal{F}_{\text{com}}$. Thus, if Sim does not abort, the output of the honest receiver $\mathcal{R}$ are identical in the real and ideal worlds, and so are the outputs of the environment $\mathcal{Z}$.

It remains to argue that if Sim aborts, then with all but negligible probability so would an honest $\mathcal{R}$ running $\mathbf{Prot}_{\mathsf{com}}$:

- Note that although the environment $\mathcal{Z}$ can query $\widehat{\mathcal{G}}_{\text{att}}$ through any dummy honest party, the sid' in the query must be different from the challenge sid. Therefore $\mathcal{Z}$ is not able to obtain any signatures pertaining to the challenge sid from $\widehat{\mathcal{G}}_{\text{att}}$ through an honest dummy party. $\mathcal{Z}$ can query $\widehat{\mathcal{G}}_{\text{att}}$ with any forged sid through the adversary $\mathcal{A}$ — however these queries are observable to Sim, and with all but negligible probability a different eid_C is generated on each enclave

installation. Thus, if the signature scheme is secure, it is easy to see that the probability of $\mathcal{R}$'s first verification succeeding, yet Sim aborting with a sig-failure1, is negligible.

- By a similar argument, conditioned on Sim not aborting with output sig-failure1, the probability of $\mathcal{R}$'s second signature verification succeeding but Sim aborting with output sig-failure2 is negligible.
- Conditioned on Sim not aborting with a sig-failure, the only way for $\mathcal{A}$ to open to a different value b' than the one extracted by Sim is by calling $\mathsf{prog}_{\mathsf{com}}$ with an input x satisfying $\mathsf{owf}(x) = c$, where c is the challenge sent by $\mathcal{R}$ to $\mathcal{A}$. By the security of the one-way function, the probability of this event is negligible.

Receiver is corrupt: When $\mathcal{A}$ sends (eid_R, c, σ) to $\mathcal{C}$, the simulator aborts and outputs sig-failure in the following cases:

- $\mathcal{A}$ never invoked $\widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathsf{owf}})$ with eid_R as response.
- (c, σ) was never output by $\widehat{\mathcal{G}}_{\mathrm{att}}$ on a valid call $\widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_R, (\text{“trapdoor”}, x))$ from $\mathcal{A}$.

Otherwise, Sim performs a reverse lookup of the value x submitted by $\mathcal{A}$ during a previous $\widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_R, x)$ query where the response was (c, σ). The simulator Sim records that x. Again, note that this x must exist and be unique, conditioned on Sim not aborting. When $\mathcal{F}_{\mathrm{com}}$ notifies Sim that $\mathcal{C}$ committed in the ideal world, Sim calls $eid_C := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathsf{com}})$ followed by $(c', \sigma') := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_C, (\text{“commit”}, 0, c))$ where c is from the message received earlier from $\mathcal{A}$. Sim sends (eid_C, c', σ') to $\mathcal{A}$ on behalf of the simulated $\mathcal{C}$. Finally, upon receiving the opening of the committed bit b^* from $\mathcal{F}_{\mathrm{com}}$, the simulator equivocates the commitment by calling $(b^*, \sigma^*) := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_C, (\text{“open”}, x^*, b^*))$. It now sends (eid_C, b^*, σ^*) to $\mathcal{A}$ on behalf of the simulated $\mathcal{C}$.

Indistinguishability: Consider the following sequence of hybrids from the real-world to the ideal-world executions.

- **Real execution.** Here $\mathcal{Z}$ interacts with $\mathcal{A}$ and an honest committer in the real-world protocol, with Sim perfectly emulating $\mathcal{C}$, while observing and forwarding messages between $\mathcal{A}$ and $\widehat{\mathcal{G}}_{\mathrm{att}}$.
- **Hybrid 1.** This is the same as the real execution, except that Sim aborts if the sig-failure event defined above occurs but $\mathcal{C}$'s signature verification for the obtained (eid_R, c, σ) would succeed. Otherwise, Sim records the pre-image x sent by $\mathcal{A}$ to $\mathsf{prog}_{\mathsf{owf}}$.

  Similarly to previous cases, the probability of a successful signature verification if Sim aborts with sig-failure can trivially be shown to be negligible under the assumption that the signature scheme is secure. Thus, no p.p.t. algorithms $(\mathcal{A}, \mathcal{Z})$ can distinguish Hybrid 1 from the real execution.
- **Hybrid 2.** The same as Hybrid 1, except that Sim always sends $(\text{“commit”}, 0, c)$ to the installed enclave in $\widehat{\mathcal{G}}_{\mathrm{att}}$. When later asked to decommit to b, Sim sends $(\text{“open”}, x, b)$ to the $\mathsf{prog}_{\mathsf{com}}$, where x is the pre-image recorded by Sim in Hybrid 1.

  As the output of $\mathsf{prog}_{\mathsf{com}}$ and the attestation σ produced by $\widehat{\mathcal{G}}_{\mathrm{att}}$ are independent of the committed value b (given that the caller knows a pre-image of c under owf), the view of $\mathcal{A}$ is identically distributed in Hybrid 2 and Hybrid 1.

$\mathsf{prog}_{\mathsf{owf}}$

On input ("trapdoor", τ): return $\mathsf{owf}(\tau)$

$\mathsf{prog}_{\mathsf{zkp}}[R]$

On input ("prove", $\mathsf{stmt}, w, c, \tau$):
  if $\mathsf{owf}(\tau) = c$, return (stmt, c)
  else if $R(\mathsf{stmt}, w) = 1$, return (stmt, c)

$\mathbf{Prot}_{\mathsf{zkp}}[sid, \mathcal{P}, \mathcal{V}, R]$

Verifier $\mathcal{V}$:

On input "verify" from $\mathcal{Z}$:
  let $eid_V := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathsf{owf}})$
  let $\tau \xleftarrow{\$} \{0,1\}^\lambda$, let $(c, \sigma) := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_V, (\text{"trapdoor"}, \tau))$
  send (eid_V, c, σ) to $\mathcal{P}$, await $(eid_P, \mathsf{stmt}, \sigma')$ from $\mathcal{P}$
  assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((sid, eid_P, \mathsf{prog}_{\mathsf{zkp}}[R], \mathsf{stmt}, c), \sigma')$
  output stmt

Prover $\mathcal{P}$:

On input ("prove", stmt, w) from $\mathcal{Z}$:
  await (eid_V, c, σ) from $\mathcal{V}$ and assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((sid, eid_V, \mathsf{prog}_{\mathsf{owf}}, c), \sigma)$
  let $eid_P := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{install}(sid, \mathsf{prog}_{\mathsf{zkp}}[R])$, let $(\_, \_, \sigma) := \widehat{\mathcal{G}}_{\mathrm{att}}.\mathtt{resume}(eid_P, (\text{"prove"}, \mathsf{stmt}, w, c, \bot))$
  send $(eid_P, \mathsf{stmt}, \sigma)$ to $\mathcal{V}$, output "okay"

Figure 14: Composable zero-knowledge proofs: both prover and verifier have secure processors. R denotes the NP relation.

Finally, it remains to observe that Hybrid 2 is indistinguishable from an ideal world execution with Sim and $\mathcal{F}_{\mathrm{com}}$.

Committer and receiver are honest: As $\mathcal{C}$ and $\mathcal{R}$ communicate over secure channels, simulating the view of the adversary $\mathcal{A}$ is trivial: when Sim is notified by $\mathcal{F}_{\mathrm{com}}$ that $\mathcal{C}$ committed, Sim emulates an interaction between $\mathcal{R}$ and $\mathcal{C}$ over a secure channel in the presence of $\mathcal{A}$ (note that the transmitted messages are of fixed length). If $\mathcal{A}$ drops any message sent between $\mathcal{C}$ and $\mathcal{R}$, Sim aborts. Sim simply forwards any messages between $\mathcal{A}$ and $\widehat{\mathcal{G}}_{\mathrm{att}}$.

□

8.3 Composable Zero-Knowledge Proofs with Transparent Enclaves

Using similar techniques, we can also realize UC-secure zero-knowledge proofs with transparent enclaves, assuming that the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ both have secure processors. We present the formal protocol in Figure 14 and briefly describe the intuition below.

Informally speaking, the protocol works as follows.

1. First, the verifier $\mathcal{V}$ will choose a high-entropy trapdoor τ and submit τ to its enclave. The enclave will compute $c := \mathsf{owf}(\tau)$ and return an attestation σ on c. As we shall see, this trapdoor τ will allow the simulator to program the enclave and equivocate when the verifier $\mathcal{V}$ is corrupt.
2. Now, the verifier $\mathcal{V}$ will send (c, σ) to the prover.

3. The prover $\mathcal{P}$ verifies the attestation, and then sends the tuple $(\mathsf{stmt}, w, c, \perp)$ to its enclave. In the real-world execution, $\mathcal{P}$'s enclave program (denoted $\mathsf{prog}_{\mathrm{zkp}}$) checks to see if w is a valid witness for the statement stmt. If so, $\mathcal{P}$'s enclave will return $(\mathsf{stmt}, c, \sigma')$ where σ' is an attestation on the pair (stmt, c). The prover $\mathcal{P}$ now sends (stmt, σ') to the verifier $\mathcal{V}$.
4. The verifier $\mathcal{V}$ verifies the attestation σ' using the value c it has sent the prover $\mathcal{P}$ earlier — note that this ensures that $\mathcal{P}$ has passed the correct c value to its enclave. If the verification succeeds, $\mathcal{V}$ accepts the statement stmt.

Extraction. When the prover $\mathcal{P}$ is corrupt, the simulator must extract the witness w. This is easy since for the the corrupt prover $\mathcal{P}$ to convince the honest verifier $\mathcal{V}$, $\mathcal{P}$ must submit a valid tuple $(\text{“prove”}, \mathsf{stmt}, w, c, \tau)$ to $\widehat{\mathcal{G}}_{\mathrm{att}}$, and the simulator can capture this message. We stress that when the corrupt prover $\mathcal{P}$ submits this message $(\text{“prove”}, \mathsf{stmt}, w, c, \tau)$, if c is the correct value that the simulator has sent $\mathcal{P}$ earlier, it cannot be the case that τ is a correct trapdoor for c, since otherwise we would be able to leverage $\mathcal{P}$ to break the one-way function owf.

Equivocation. We now explain a backdoor in the prover $\mathcal{P}$'s enclave program (denoted $\mathsf{prog}_{\mathrm{zkp}}$), and explain how to make use of this to allow the simulator to equivocate.

This backdoor works as follows: if $\mathcal{P}$ supplies a valid trapdoor τ such that $\mathsf{owf}(\tau) = c$, then the enclave program will sign any statement stmt provided by $\mathcal{P}$ regardless of whether the witness w is valid. In the real-world execution, if the verifier $\mathcal{V}$ is honest, it will check that $\mathcal{P}$ has passed the correct value c to its enclave. Since $\mathcal{V}$ never discloses its private trapdoor to anyone and that the one-way function is secure, the real-world prover $\mathcal{P}$ should never be able to come up with the correct trapdoor; and thus the real-world prover $\mathcal{P}$ must supply a correct witness to get its enclave to sign the statement stmt. This ensures soundness in the real world.

In the simulation, however, if the verifier $\mathcal{V}$ is corrupt, the simulator must be able to get $\widehat{\mathcal{G}}_{\mathrm{att}}$ to sign the statement stmt without knowning the witness. This is where the simulator must invoke this backdoor. More specifically, when the corrupt verifier $\mathcal{V}$ submits the trapdoor τ to its enclave, the simulator can capture this message and extract the trapdoor τ. This enables the simulator to later invoke the backdoor and program the enclave.

Theorem 19 (Composable zero-knowledge proofs with transparent enclaves)**.** *Assume that* owf *is a secure one-way function, and that* Σ *is secure, the protocol described in Figure 14 UC-realizes* $\mathcal{F}_{zkp}[sid, \mathcal{P}, \mathcal{V}]$ *where* $\mathcal{P}$ *denotes the prover, and* $\mathcal{V}$ *denotes the verifier.*

Proof. Notice that the zero-knowledge proof protocol (Figure 14) is very similar to the commitment protocol (Figure 13). The most notable difference is that commitment has two phases, namely “commit” and “open”; but a zero-knowledge proof only has a single phase. The proof of this theorem is thus very similar to that of Theorem 18 — we omit the full proofs here to avoid being repetitive. □

8.4 Non-Anonymous Attestation

While some secure processors such as Intel SGX rely on anonymous attestation, others such as older versions of TPM rely on non-anonymous attestation. One typical realization is for the manufacturer to sign a certificate for the secure processor's long-term public key, and the corresponding secret key is embedded in non-volatile memory inside the secure processor, such that the secure processor

can sign attestations with it. In general, such a signature chain can be thought of as using the manufacturer's public key mpk to sign messages prefixed by the platform's identity.

It is not hard to see that our protocols for a single secure processor that leverage $\mathcal{G}_{\text{acrs}}$ and witness indistinguishable proofs (see Sections 6 and 7.5) can easily be adapted to work with non-anonymous attestation — the only modification needed is to add the platform's identity (denoted $\mathcal{P}$) as an extra witness in the witness-indistinguishable proof, and prove that either the encrypted witness is the pair $(\mathcal{P}, \sigma)$ such that σ is a valid attestation on $\mathcal{P}||\mathsf{msg}$; or the encrypted witness is the receiver's identity key.

However, note that our protocols that rely on all parties to have a secure processor no longer work in the case of non-anonymous attestation. Specifically, these protocols (see Sections 5.2 and 7.6) send attestations around in the clear. In the case of anonymous attestation and when secure processors are omnipresent, sending attestations around cannot implicate an honest party of participation; however, in the case of non-anonymous attestation, since the attestation now binds to the party's identifier, sending signatures around in the clear would lead to non-deniability.

Acknowledgments

We thank Elette Boyle, Kai-Min Chung, Victor Costan, Srini Devadas, Ari Juels, Andrew Miller, Dawn Song, and Fan Zhang for helpful and supportive discussions. This work is supported in part by NSF grants CNS-1217821, CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, AFOSR Award FA9550-15-1-0262, an Office of Naval Research Young Investigator Program Award, a Microsoft Faculty Fellowship, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, and a VMWare Research Award. This work was done in part while a subset of the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant CNS-1523467. The second author would like to thank Adrian Perrig and Leendert van Doorn for many helpful discussions on trusted hardware earlier in her research.

References

[1] Intel SGX for dummies. `https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx`.

[2] Trusted computing group. `http://www.trustedcomputinggroup.org/`.

[3] Martín Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. In *Theoretical Aspects of Computer Software*, pages 82–94, 2001.

[4] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 20(3):395, 2007.

[5] Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness of formal encryption in the presence of key-cycles. In *ESORICS*, pages 374–396, 2005.

[6] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3(4):18–24, 2004.

[7] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.

[8] ARM Limited. *ARM Security Technology Building a Secure System using TrustZone® Technology*, Apr 2009. Reference no. PRD29-GENC-009492C.

[9] Gilad Asharov, Amos Beimel, Nikolaos Makriyannis, and Eran Omri. Complete characterization of fairness in secure two-party computation of boolean functions. In *Theory of Cryptography Conference (TCC)*, pages 199–228, 2015.

[10] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A universally composable cryptographic library. *IACR Cryptology ePrint Archive*, 2003:15, 2003.

[11] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, 2001.

[12] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *IEEE European Symposium on Security and Privacy*, pages 245–260, 2016.

[13] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.

[14] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX Security*, 2006.

[15] Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, Yael Tauman Kalai, and Guy N. Rothblum. Program obfuscation with leaky hardware. In *ASIACRYPT*, 2011.

[16] Florian Bohl and Dominique Unruh. Symbolic universal composability. In *IEEE Computer Security Foundations Symposium*, pages 257–271, 2013.

[17] Dan Boneh and Moni Naor. Timed commitments. In *CRYPTO*, 2000.

[18] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS*, 2004.

[19] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing. *IACR Cryptology ePrint Archive*, 2009:95, 2009.

[20] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.

[21] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[22] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*. 2007.

[23] Ran Canetti and Marc Fischlin. Universally composable commitments. In *Advances in Cryptology (CRYPTO)*, pages 19–40, 2001.

[24] Ran Canetti and Jonathan Herzog. Universally composable symbolic security analysis. *J. Cryptology*, 24(1):83–147, 2011.

[25] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical uc security with a global random oracle. In *CCS'14*, pages 597–608, 2014.

[26] Ran Canetti and Tal Rabin. Universal composition with joint state. In *CRYPTO*, 2003.

[27] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global pki. In *PKC'16*, pages 265–296. Springer, 2016.

[28] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.

[29] Chen Chen, Himanshu Raj, Stefan Saroiu, and Alec Wolman. cTPM: A cloud TPM for cross-device trusted applications. In *NSDI*, 2014.

[30] Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. Functional encryption from (small) hardware tokens. In *Asiacrypt*, 2013.

[31] Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. Functional encryption from (small) hardware tokens. In *ASIACRYPT*, 2013.

[32] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC'86*, pages 364–369, 1986.

[33] Victor Costan and Srini Devadas. Intel SGX explained. Manuscript, 2015.

[34] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. Cryptology ePrint Archive, Report 2015/564, 2015. `http://eprint.iacr.org/`.

[35] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling stronger privacy in MapReduce computation. In *USENIX Security*, 2015.

[36] Nico Döttling, Thilo Mie, Jörn Müller-Quade, and Tobias Nilges. Basing obfuscation on simple tamper-proof hardware assumptions. *IACR Cryptology ePrint Archive*, 2011:675, 2011.

[37] Nico Döttling, Thilo Mie, Jörn Müller-Quade, and Tobias Nilges. Implementing resettable uc-functionalities with untrusted tamper-proof hardware-tokens. In *TCC*, 2013.

[38] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6), June 1985.

[39] Andrew Ferraiuolo, Yao Wang, Rui Xu, Danfeng Zhang, Andrew Myers, and G. Edward Suh. Full-processor timing channel protection with applications to secure hardware compartments. 2015.

[40] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[41] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[42] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[43] Juan Garay, Philip MacKenzie, Manoj Prabhakaran, and Ke Yang. Resource fairness and composability of cryptographic protocols. In *TCC*, 2006.

[44] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. Physical key extraction attacks on pcs. *Commun. ACM*, 59(6):70–79, May 2016.

[45] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *CRYPTO*, 2008.

[46] Dov Gordon and Jonathan Katz. Complete fairness in multi-party computation without an honest majority. In *TCC*, 2009.

[47] S. Dov Gordon, Carmit Hazay, Jonathan Katz, and Yehuda Lindell. Complete fairness in secure two-party computation. *J. ACM*, 58(6):24:1–24:37, December 2011.

[48] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, 2010.

[49] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin R. B. Butler, and Patrick Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *FC*, 2016.

[50] Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkitasubramaniam. Composable security in the tamper-proof hardware model under minimal complexity. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, pages 367–399, 2016.

[51] Omer Horvitz and Virgil D. Gligor. Weak key authenticity and the computational completeness of formal encryption. In *CRYPTO*, pages 530–547, 2003.

[52] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX)*, Jun 2015. Reference no. 332680-002.

[53] Romain Janvier, Yassine Lakhnech, and Laurent Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In *ESOP*, pages 172–185, 2005.

[54] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, 2007.

[55] Bernhard Kauer. Tpm reset attack. `http://www.cs.dartmouth.edu/~pkilab/sparks/`.

[56] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[57] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[58] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[59] Lorenzo Martignoni, Pongsin Poosankam, Matei Zaharia, Jun Han, Stephen McCamant, Dawn Song, Vern Paxson, Adrian Perrig, Scott Shenker, and Ion Stoica. Cloud terminal: Secure access to sensitive applications from untrusted systems. In *USENIX ATC*, 2012.

[60] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

[61] Jeremias Mechler, Jrn Mller-Quade, and Tobias Nilges. Universally composable (non-interactive) two-party computation from untrusted reusable hardware tokens. Cryptology ePrint Archive, Report 2016/615, 2016. `http://eprint.iacr.org/2016/615`.

[62] Daniele Micciancio and Bogdan Warinschi. Completeness theorems for the Abadi-Rogaway language of encrypted expressions. *J. Comput. Secur.*, 12(1):99–129, January 2004.

[63] Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference (TCC)*, 2004.

[64] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, August 2016.

[65] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *POST*, pages 53–72, 2015.

[66] Adam Petcher and Greg Morrisett. A mechanized proof of security for searchable symmetric encryption. In *CSF*, 2015.

[67] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.

[68] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. *SIGARCH Comput. Archit. News*, 42(1):67–80, February 2014.

[69] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security*, pages 175–188, 2012.

[70] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. In *IEEE S&P*, 2015.

[71] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *IEEE S&P*, 2005.

[72] Elaine Shi, Fan Zhang, Rafael Pass, Srini Devadas, Dawn Song, and Chang Liu. Systematization of knowledge: Trusted hardware: Life, the composable universe, and everything. Manuscript, 2015.

[73] Sean W. Smith and Vernon Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *Proceedings of the 3rd Conference on USENIX Workshop on Electronic Commerce - Volume 3*, WOEC'98, 1998.

[74] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, ICS '03, pages 160–171, 2003.

[75] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[76] Mike Szczys. TPM crytography cracked. `http://hackaday.com/2010/02/09/tpm-crytography-cracked/`.

[77] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.*, 34(5):168–177, November 2000.

[78] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, August 1984.

[79] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *IEEE European Symposium on Security and Privacy*, 2017.

[80] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[81] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.

[82] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *ACM CCS*, 2016.

[83] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News*, 32(5):72–84, October 2004.

Appendices

A Universal Composition Background and Conventions

In this section, we first give a brief background on the universal composition framework [21, 22, 26]. We then introduce some new UC conventions we will adopt throughout this paper.

A.1 Brief Background on the Universal Composition Framework

The UC framework allows for *modular* analysis of large programs: various subroutines can be analyzed as separate entities, the security of which is assessed independently by means of realizing some *ideal functionality* $\mathcal{F}$. The universal composabilitiy theorem then states, informally, that the security properties of a protocol that makes subroutine calls to $\mathcal{F}$ are retained if $\mathcal{F}$ is replaced by the actual program or protocol that realizes it.

At a high level, security in the UC framework consists in showing that whatever information can be learned by some network adversary $\mathcal{A}$ in a *real world* execution of some protocol π, could also have been obtained by a simulator Sim attacking an *ideal world* protocol execution, where all parties privately interact with an idealized trusted functionality $\mathcal{F}$.

The presence of arbitrary other protocols running alongside π is modeled via an *environment* $\mathcal{Z}$, which determines the inputs to all parties participating in a protocol and sees all the parties' outputs. The environment interacts with the adversary ($\mathcal{A}$ or Sim) to coordinate party corruptions. The protocol π is said to UC realize some ideal functionality $\mathcal{F}$, if for any real-world adversary $\mathcal{A}$, there exists a simulator Sim, such that no p.p.t. environment $\mathcal{Z}$ can distinguish an interaction with $\mathcal{A}$ and parties running π from an interaction with Sim and $\mathcal{F}$.

By the composition theorem [21], any protocol that makes subroutine calls to $\mathcal{F}$ retains its security properties if all calls to $\mathcal{F}$ are appropriately replaced by instances of a protocol π that UC realizes $\mathcal{F}$.

Setup functionalities. Many functionalities of interest cannot be realized in the "plain" UC framework. It is thus customary to consider *hybrid models*, in which parties get access to some ideal setup functionality.

In the basic UC framework, these setup functionalities are "local" to a particular protocol execution. The environment $\mathcal{Z}$ (which we recall models arbitrary other protocols running in the network) can only interact with the setup functionality $\mathcal{F}$ through the adversary. In this sense, the basic UC framework fails to capture composable security in the presence of globally available setup functionalities such as PKIs or trusted hardware platforms.

UC with global setup. To remedy the shortcoming of the plain UC framework, Canetti et al. [22] propose a Generalized UC model (GUC), which enables proofs of secure composition in the presence of *global* setup functionalities, that can be accessed by any party in any protocol instance in the system.

The main technical difference compared to the original UC framework, is that the environment $\mathcal{Z}$ is now allowed to interact with the setup functionality "directly", i.e., without going through $\mathcal{A}$ or Sim. The setup functionality must thus be *non-programmable*, meaning that the simulator Sim cannot select the secret state that makes up the setup functionality, as this would be detectable by the environment.

As a concrete example, if we were to model trusted hardware in the UC framework, the simulator could select the secret key used for signing attestation, and communicate the corresponding public key to $\mathcal{Z}$. In the GUC framework however, $\mathcal{Z}$ need not interact with Sim to obtain the trusted platform's public key. Any simulation must then be consistent with this global key pair, with Sim having no knowledge of the secret key.

Previous works [22, 25] have noticed that achieving security in the GUC model is remarkably non-trivial: although the global setup has to be publicly available, it must also provide some kind of hidden "trapdoor" information, that the simulator can exploit to "cheat" in a simulation.

A.2 UC Notational Conventions

In this paper, we use the term UC and GUC indistinguishably. Let $\mathcal{G}$ denote a global functionality. When we say that a $\mathcal{G}$-hybrid protocol UC-realizes a functionality, we mean that the standard UC simulation definition holds in light of the fact that $\mathcal{Z}$ can interact with $\mathcal{G}$.

Session conventions. UC assumes that the environment $\mathcal{Z}$ invokes each protocol instance with a unique session identifier sid. While earlier UC papers adopt the convention that ITIs include sid and often the party identifiers in messages, in this paper, we use a simplified notation where we simply parametrize the functionality or protocol instance with the session identifier as well as identifiers of parties involved. For example, $\mathcal{F}_{\text{outsrc}}[sid, \mathcal{C}, \mathcal{S}]$ denotes an $\mathcal{F}_{\text{outsrc}}$ instance with session identifier sid, and involving a client $\mathcal{C}$ and a server $\mathcal{S}$. Similarly, $\mathbf{Prot}_{\text{outsrc}}[sid, \mathcal{C}, \mathcal{S}]$ denotes a protocol instance with session identifier sid, and involving a client $\mathcal{C}$ and a server $\mathcal{S}$. In this way, both the sid and the party identifiers can be referenced by the code of the ITIs. Note that this convention only works when the sid and party identifiers can be statically determined (as opposed to dynamically or at run-time), which is the case throughout this paper.

Reentrant and non-reentrant activation points. In this paper, all reentrant activation points for ITIs are colored blue and followed by an asterisk*, and all non-reentrant activation points for ITIs are colored green. A reentrant activation point can be invoked multiple times. A non-reentrant activation point can only be invoked once; and for all future invocations the ITI would do nothing and immediately return $\perp$.

Assertions. When we write the code for ITIs, we often use assertions. When assertions fail, the ITI immediately returns $\perp$. When enclave programs have an assertion failure, we assume that $\mathcal{G}_{\text{att}}$ simply returns $\perp$.

Secure channels. In our paper, we use the notation send to denote a UC-secure secure channel, realized with the standard secure channel functionality denoted $\mathcal{F}_{\text{sc}}$. The secure channel functionality can be realized from a global PKI using Diffie-Hellman key exchange and authenticated encryption for instance [27].

A.3 Multi-Party Computation

Useful functionalities. We define a few well-known ideal functionalities, including multi-party computation, commitment, and zero-knowledge proofs. The definitions are given in Figure 15,

$\mathcal{F}_{\text{mpc}}[sid, f, \mathcal{P}_1, \ldots, \mathcal{P}_n]$

On receive* x_i from $\mathcal{P}_i$:
notify $\mathcal{A}$ of $\mathcal{P}_i$, store x_i and ignore further messages from $\mathcal{P}_i$
if an x_i was received from all $\mathcal{P}_i$, let $y := f(x_1, \ldots, x_n)$
generate delayed private outputs y for each party $\mathcal{P}_1, \ldots, \mathcal{P}_n$

Figure 15: Ideal multiparty computation functionality.

$\mathcal{F}_{\text{com}}[sid, \mathcal{C}, \mathcal{R}]$

On receive ("commit", x) from $\mathcal{C}$:
store x, generate a public delayed message "receipt" to $\mathcal{R}$.

On receive "open" from $\mathcal{C}$:
assert some x was stored, generate a private delayed message ("open", x) to $\mathcal{R}$.

Figure 16: Ideal two-party commitment functionality.

$\mathcal{F}_{\text{zkp}}[sid, \mathcal{P}, \mathcal{V}, R]$

On receive ("prove", stmt, w) from $\mathcal{P}$:
if $R(\mathsf{stmt}, w) = 1$, generate a private delayed message stmt to $\mathcal{V}$

Figure 17: Ideal zero-knowledge proof functionality.

Figure 16, and Figure 17 respectively. Whenever applicable, we assume for simplicity that the input and output lengths are fixed and known publicly in advance.

Sequentially and universally composable multi-party computation. We define two notions of multi-party computation: sequentially composable MPC and UC-secure MPC. If a protocol is UC-secure, then it is sequentially composable. We will use the weaker notion of security, i.e., sequentially composable MPC for our lower bounds (and this makes our lower bound results stronger), but the stronger security notion for our constructions.

Definition 2 (Sequentially composable multi-party computation [20])**.** *We say that protocol* Π *realizes* $\mathcal{F}$*, if for any p.p.t. adversary* $\mathcal{A}$*, there exists a p.p.t. simulator* $\mathcal{S}$*,*

$$\{\text{IDEAL}^{\mathcal{F},\mathcal{S}}(\lambda, x_1, \ldots, x_n, z)\}_{x_1,\ldots,x_n,z} \stackrel{c}{\equiv} \{\text{EXEC}^{\Pi,\mathcal{A}}(\lambda, x_1, \ldots, x_n, z)\}_{x_1,\ldots,x_n,z}$$

where $x_1, \ldots, x_n$ *denotes the* n *parties' respective inputs, and* z *denotes an auxiliary advice string provided to the adversary.*

In the above, the notations IDEAL and EXEC denote the random variable consisting of the honest parties' outputs as well as the protocol transcripts as viewed by the corrupt parties.

Definition 3 (Universally composable MPC [21])**.** *We say that protocol* Π *UC-realizes* $\mathcal{F}$*, if for any p.p.t. adversary* $\mathcal{A}$*, there exists a p.p.t. simulator* $\mathcal{S}$*, such that for any p.p.t. environment* $\mathcal{Z}$*,*

$$\text{IDEAL}^{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) \stackrel{c}{\equiv} \text{EXEC}^{\Pi,\mathcal{A},\mathcal{Z}}(\lambda)$$

In the above, the environment $\mathcal{Z}$ is allowed to adaptively choose inputs for all parties, and communicate with the adversary in arbitrary manners. The notations IDEAL and EXEC denote the views of the environment $\mathcal{Z}$ in the executions, including all parties' inputs and outputs, as well as any protocol transcript as viewed by the adversary.

Note that the main difference between the two definitions is the following: for sequential composition, the simulator $\mathcal{S}$ can depend on the adversary $\mathcal{A}$; whereas in universal composable security, the simulator $\mathcal{S}$ must work for all environment $\mathcal{Z}$ (and the adversary $\mathcal{A}$ can be considered dummy, i.e., only pass messages between $\mathcal{Z}$ and the honest parties).

A.4 Preliminaries on Zero-Knowledge Proofs

In the remainder of this section, $f(\lambda) \approx g(\lambda)$ means that there exists a negligible function $\nu(\lambda)$ such that $|f(\lambda) - g(\lambda)| < \nu(\lambda)$.

A non-interactive proof system henceforth denoted NIWI for an NP language $\mathcal{L}$ consists of the following algorithms:

- $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{L})$, also written as $\mathsf{crs} \leftarrow \mathsf{KeyGen}_{\mathsf{NIWI}}(1^\lambda, \mathcal{L})$: Takes in a security parameter λ, a description of the language $\mathcal{L}$, and generates a common reference string crs. In this paper, we use a global crs, which is part of our global setup $\mathcal{G}_{\text{acrs}}$.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w)$: Takes in crs, a statement stmt, a witness w such that $(\mathsf{stmt}, w) \in \mathcal{L}$, and produces a proof π.
- $b \leftarrow \mathsf{Ver}(\mathsf{crs}, \mathsf{stmt}, \pi)$: Takes in a crs, a statement stmt, and a proof π, and outputs 0 or 1, denoting accept or reject.

Perfect completeness. A non-interactive proof system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any $(\mathsf{stmt}, w) \in R$, we have that

$$\Pr\left[\ \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{L}),\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w) : \mathsf{Ver}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1\ \right] = 1$$

Computational soundness. A non-interactive proof system for the language $\mathcal{L}$ is said to be computationally sound, if for all p.p.t. adversaries $\mathcal{A}$,

$$\Pr\left[\ \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{L}), (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) : (\mathsf{Ver}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1) \wedge (\mathsf{stmt} \notin \mathcal{L})\ \right] \approx 0$$

Witness indistinguishability. A non-interactive proof system for the language $\mathcal{L}$ is said to be computationally sound, if for all p.p.t. adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{L}),\\ (\mathsf{stmt}, w_0, w_1) \leftarrow \mathcal{A}(\mathsf{crs}),\\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w_0) :\\ (\mathsf{stmt}, w_0) \in \mathcal{L} \wedge (\mathsf{stmt}, w_1) \in \mathcal{L} \wedge \mathcal{A}(\pi) = 1\end{array}\right] \approx \Pr\left[\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{L}),\\ (\mathsf{stmt}, w_0, w_1) \leftarrow \mathcal{A}(\mathsf{crs}),\\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w_1) :\\ (\mathsf{stmt}, w_0) \in \mathcal{L} \wedge (\mathsf{stmt}, w_1) \in \mathcal{L} \wedge \mathcal{A}(\pi) = 1\end{array}\right]$$

$\mathcal{F}_{\text{outsrc}}[sid, \mathcal{C}, \mathcal{S}]$

On receive* ("compute", f, x) from $\mathcal{C}$:
  let $y := f(x)$
  send $(|f| + |x|, |y|)$ to $\mathcal{S}$ and $\mathcal{A}$
  generate a delayed private output y to $\mathcal{C}$

Figure 18: The ideal secure outsourcing functionality. The server and adversary learn nothing more than the size of the client's inputs and outputs

Note that although we define non-interactive witness indistinguishable proofs in the global common reference string model for ease of exposition, our protocol and proof (for when only a single party has a secure processor) still work if we instead adopted interactive versions that do not require a global common reference string — nonetheless our protocols for a single secure processor would require the use of $\mathcal{G}_{\text{acrs}}$ elsewhere to circumvent a theoretical impossibility that we show.

B Warmup: Secure Outsourcing from $\mathcal{G}_{\text{att}}$

To illustrate the usage of the $\mathcal{G}_{\text{att}}$ setup assumption to achieve formal composable security, we begin by considering a very simple outsourcing application. The ideal functionality we wish to achieve is denoted $\mathcal{F}_{\text{outsrc}}$ and described in Figure 18.

In Figure 19 we show a simple protocol between a client $\mathcal{C}$ and a server $\mathcal{S}$ to realize $\mathcal{F}_{\text{outsrc}}$. The server is in possession of a trusted hardware platform and initializes an enclave running the public program $\mathsf{prog}_{\text{outsrc}}$.

Theorem 20 (Secure outsourcing from $\mathcal{G}_{\text{att}}$)**.** *Assume that the signature scheme Σ is existentially unforgeable under chosen message attacks, the Decisional Diffie-Hellman assumption holds in the algebraic group adopted, the authenticated encryption scheme AE is perfectly correct and satisfies the standard notions of INT-CTXT and semantic security. Then, the $\mathcal{G}_{att}$ – hybrid protocol $\mathbf{Prot}_{outsrc}$ UC-realizes $\mathcal{F}_{outsrc}$ when the client $\mathcal{C}$ is honest, and the server $\mathcal{S}$ is a static, malicious adversary.*

Proof. We now prove Theorem 20, i.e., that $\mathbf{Prot}_{\text{outsrc}}$ securely realizes $\mathcal{F}_{\text{outsrc}}$. We consider two cases: when the client is honest and server is corrupt; and when both the client and server are honest.

Honest client and corrupt server. We first describe an ideal-world simulator Sim, and then show that no p.p.t. environment $\mathcal{Z}$ can distinguish the ideal-world and real-world executions.

- Unless noted otherwise below, any communication between $\mathcal{Z}$ and $\mathcal{A}$ or between $\mathcal{A}$ and $\mathcal{G}_{\text{att}}$ is simply forwarded by Sim.
- The simulator Sim starts by emulating the setup of a secure channel between $\mathcal{C}$ and $\mathcal{G}_{\text{att}}$. Sim sends ("keyex", g^a) to $\mathcal{A}$ (that controls the corrupted $\mathcal{S}$) for a randomly chosen a.
- When Sim receives a tuple (eid, g^b, σ) from $\mathcal{A}$, Sim aborts outputting $\mathsf{sig\text{-}failure}$ if σ would be validated by a honest $\mathcal{C}$, yet Sim has not recorded the following $\mathcal{A} \Leftrightarrow \mathcal{G}_{\text{att}}$ communication:

$\mathsf{prog}_{\text{outsrc}}$

On input $(\text{“keyex”}, g^a)$: let $b \leftarrow\!\!\$\, \mathbb{Z}_p$, store $\mathsf{sk} := (g^a)^b$; return (g^a, g^b)

On input* $(\text{“compute”}, \mathsf{ct})$:
let $(f, x) := \mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct})$
assert decryption success, ct not seen before
let $y := f(x)$ and return $\mathsf{ct}_{\text{out}} := \mathsf{AE.Enc}_{\mathsf{sk}}(y)$

$\mathbf{Prot}_{\text{outsrc}}[sid, \mathcal{C}, \mathcal{S}]$

Server $\mathcal{S}$:

On receive $(\text{“keyex”}, g^a)$ from $\mathcal{C}$:
let $eid := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{outsrc}})$
let $((g^a, g^b), \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“keyex”}, g^a))$ and send (eid, g^b, σ) to $\mathcal{C}$

On receive* $(\text{“compute”}, \mathsf{ct})$ from $\mathcal{C}$:
let $(\mathsf{ct}_{\text{out}}, \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, \mathsf{ct}))$ and send ct_{out} to $\mathcal{C}$

Client $\mathcal{C}$:

On initialize:
let $a \leftarrow\!\!\$\, \mathbb{Z}_p$, $\mathsf{mpk} := \mathcal{G}_{\text{att}}.\texttt{getpk}()$
send $(\text{“keyex”}, g^a)$ to $\mathcal{S}$, await (eid, g^b, σ) from $\mathcal{S}$
assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((sid, eid, \mathsf{prog}_{\text{outsrc}}, (g^a, g^b)), \sigma)$
let $\mathsf{sk} := (g^b)^a$

On receive* $(\text{“compute”}, f, x)$ from $\mathcal{Z}$:
let $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f, x)$ and send $(\text{“compute”}, \mathsf{ct})$ to $\mathcal{S}$, await ct_{out}
let $y := \mathsf{AE.Dec}_{\mathsf{sk}}(\mathsf{ct}_{\text{out}})$ and assert decryption success and ct_{out} not seen before
output y

Figure 19: A protocol $\mathbf{Prot}_{\text{outsrc}}$ that UC-realizes the secure outsourcing functionality $\mathcal{F}_{\text{outsrc}}$. The public group parameters (g, p) are hardcoded into $\mathsf{prog}_{\text{outsrc}}$.

- $eid := \mathcal{G}_{\text{att}}.\texttt{install}(sid, \mathsf{prog}_{\text{outsrc}})$;
- $((g^a, g^b), \sigma) := \mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“keyex”}, g^a))$

Else, Sim computes $\mathsf{sk} = g^{ab}$.

- When Sim receives a message $(|f + x|, |y|)$ from $\mathcal{F}_{\text{outsrc}}$, it proceeds as follows: Sim sends $(\text{“compute”}, \mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}((f_0, x_0)))$ to $\mathcal{A}$ where f_0 is some canonical function and x_0 some canonical input. For simplicity, we assume that functions, inputs and outputs computed by $\mathcal{F}_{\text{outsrc}}$ are of fixed size.

- Then, Sim waits to receive ct_{out} from $\mathcal{A}$. If ct_{out} was not the result of a previous $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, \mathsf{ct}))$ call but ct_{out} successfully decrypts under sk, the simulator aborts outputting $\mathsf{authenc\text{-}failure}$. Otherwise, Sim allows $\mathcal{F}_{\text{outsrc}}$ to deliver y to $\mathcal{C}$ in the ideal world.

We now prove the indistinguishability of the real-world and ideal-world executions through a sequence of hybrids.

Claim 27. *Assume that the signature scheme Σ is secure, except with negligible probability, the simulated execution does not abort outputting* sig-failure*.*

Proof. Straightforward reduction to the security of the digital signature scheme Σ. □

Hybrid 1. Identical to the simulated execution, but the secret key $\mathsf{sk} = g^{ab}$ shared between $\mathcal{C}$ and $\mathcal{G}_{\text{att}}$ is replaced with a random element from the appropriate domain.

Claim 28. *Assume that the DDH assumption holds, then Hybrid 1 is computationally indistinguishable from the simulated execution.*

Proof. Straightforward by reduction to the DDH assumption. □

Claim 29. *Assume that* AE *satisfies INT-CTXT security. It holds that in Hybrid 1,* authenc-failure *does not happen except with negligible probability.*

Proof. Straightforward by reduction to the INT-CTXT security of authenticated encryption. If $\mathcal{A}$ makes a $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, \mathsf{ct}'))$ call where ct' is not the ciphertext previously sent by Sim, either ct' is a previously seen ciphertext (causing $\mathsf{prog}_{\text{outsrc}}$ to abort, or the decryption of ct' in $\mathsf{prog}_{\text{outsrc}}$ fails with overwhelming probability.

Similarly, is the output ct_{out} sent by $\mathcal{A}$ to Sim does not come from a correct $\mathcal{G}_{\text{att}}.\texttt{resume}(eid, (\text{“compute”}, \mathsf{ct}))$ call, then either ct_{out} is a previously seen ciphertext, or $\mathcal{C}$'s decryption would fails with overwhelming probability. □

Hybrid 2. Instead of sending $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f_0, x_0)$ to $\mathcal{A}$, the simulator now sends $\mathsf{ct} := \mathsf{AE.Enc}_{\mathsf{sk}}(f, x)$ where f and x are the honest client's true inputs.

Claim 30. *Assume that* AE *is semantically secure, Hybrid 2 is computationally indistinguishable from Hybrid 1.*

Proof. Straightforward reduction to the semantic security of authenticated encryption. □

Hybrid 3. Now instead of using a random key between $\mathcal{C}$ and $\mathcal{G}_{\text{att}}$, we switch back to using the real key g^{ab}.

Claim 31. *Assume that the DDH assumption holds, then Hybrid 3 is computationally indistinguishable from Hybrid 2.*

Proof. Straightforward by reduction to the DDH assumption. □

Finally, observe that conditioned on the simulator not aborting and AE being perfectly correct, Hybrid 3 is identically distributed as the real execution.

Honest client and server. The setting where both parties are honest is trivial, as all communication between $\mathcal{S}$ and $\mathcal{C}$ is assumed to occur over secure channels.

While most of the information sent between $\mathcal{S}$ and $\mathcal{C}$ could be easily simulated in the presence of authenticated channels, the attestations σ produced by $\mathcal{G}_{\text{att}}$ are more problematic: If both parties are honest, the simulator cannot obtain valid signatures from $\mathcal{G}_{\text{att}}$ (assuming the adversary has corrupted no party $\mathcal{P} \in \texttt{reg}$). If the protocol was using authenticated channels, it would be necessary to model the fact that protocol execution can "leak" valid signatures from $\mathcal{G}_{\text{att}}$ to the adversary. This is a *deniability issue*. The signature is proof that some party $\mathcal{P} \in \texttt{reg}$ was involved in the protocol.

When introducing protocols for general two-party computation where a single party has access to trusted hardware, we show how to relax our $\mathcal{G}_{\text{att}}$ assumption to resolve this deniability issue.

□