Formal Abstractions for Attested Execution Secure Processors

Rafael Pass[1], Elaine Shi[2], Florian Tramèr[3]

[1]CornellTech, [2]Cornell, [3]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.

1 Introduction

The science of cybersecurity is founded atop one fundamental guiding principle, that is, to minimize a system's Trusted Computing Base (TCB) [70]. 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 [11, 28, 50, 61, 62], mobile security [60], web security, to cryptocurrencies [74]. In the past three decades, numerous secure processors have been proposed and demonstrated by both academia and industry [6, 22, 27, 31, 32, 48, 49, 51, 66, 73]; and several have been commercialized, including the well-known Trusted Platform Modules (TPMs) [1], Arm's TrustZone [5, 7], and others. Notably, Intel's recent release of its new x86 security extensions called SGX [6, 26, 51] 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 [1,6,27,51,67,69]. 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 [6, 51] 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. [64] 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 [11, 12, 23, 28, 50, 55, 59, 61–63]. 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 [1, 6, 26, 51]), 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 [31, 57, 58, 73], 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 [41, 55, 62] 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 [18] (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 [18], 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 2.5). 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 [25]. 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 [8, 38, 39]. 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" [71], 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 [6, 22, 27, 31, 32, 48, 49, 51, 67, 73], how to apply them in applications [11, 12, 23, 28, 50, 55, 59, 61–63], side-channels and other attacks [36, 46, 47, 68, 72, 75] and protection against such

attacks [32, 49, 73, 75]. 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 [31, 57, 58, 73], 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 [27]; others additionally defend against physical attackers [32–34, 49]; others defend against a restricted class of software and/or physical attackers that do not exploit certain side channels [1, 6, 48, 51, 66]. We refer the reader to a comprehensive systematization of knowledge paper by Shi et al. [64] 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 [27, 31–34, 49, 73, 75].

2 Technical Roadmap

2.1 Formal Modeling

Modeling choices. To enable cryptographically sound reasoning, we adopt the universal composition (UC) paradigm in our modeling [17, 18, 21]. 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 the full version of this work [56] for a more detailed overview of the UC framework in our context. 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 [2–4, 9, 13, 20, 43, 44, 53, 54].

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 [6, 51]. 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. More detailed discussions can be found in the full version [56, 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.

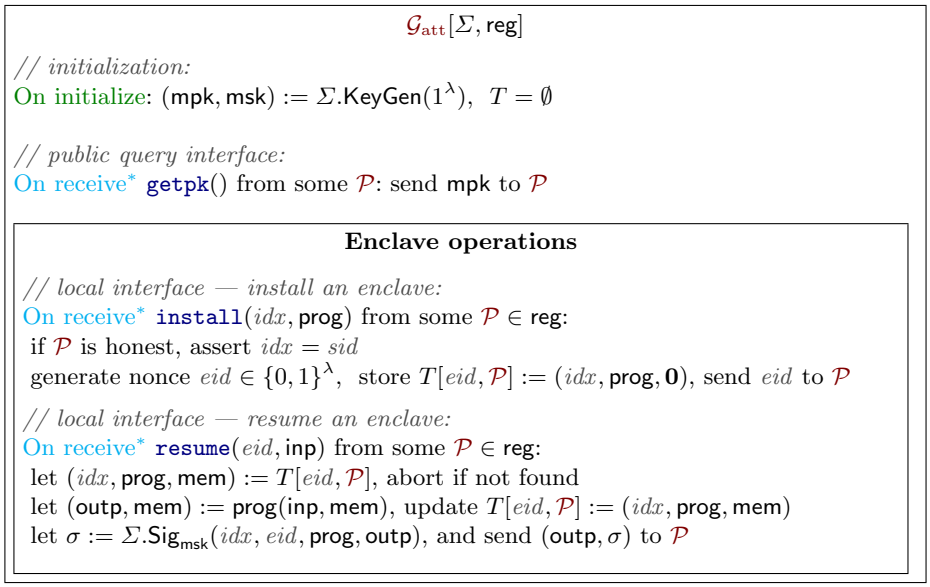

Fig. 1: A global functionality modeling an SGX-like secure processor. Blue (and starred*) activation points denote *reentrant* activation points. Green activation points are executed at most once. The enclave program prog may be probabilistic and this is important for privacy-preserving applications. 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 [15, 16] 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 [18]. 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. More detailed discussions of our modeling choices, and importantly, clarifications on how the environment $\mathcal{Z}$ interacts with $\mathcal{G}_{\text{att}}$ are deferred to our technical report [56, 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 [17,19], 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 [19]. 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 [19], 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 capture the reusability of real-world secure processors, and this justifies why we model attested execution processors as a globally shared functionality.

Unfortunately, when $\mathcal{G}_{\text{att}}$ 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 $\mathcal{G}_{\text{att}}$, 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 $\mathcal{G}_{\text{att}}$. 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

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

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

Fig. 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 hard-coded into $\mathsf{prog}_{\mathrm{2pc}}$.

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 $v = \perp$, $\mathcal{G}_{\mathrm{att}}$ 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 our full version [56, 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) [18], 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 [18], 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. We defer formal notations and proofs to [56, Section 6].

- 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

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

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

Fig. 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.

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 [18].

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 [25] 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 [8, 38, 39].

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 [14, 30, 35]. 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 $\mathbf{0}$ (see our full version [56, 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 [8, 38, 39]. We

formalize this folklore proof and also give an alternative proof in the full version of this work [56, Section 7.4].

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 the full version of this work [56, 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 [31–34, 49, 73].

Recently, Tramèr et al. [71] ask the question, what kind of interesting applications can we realize assuming that such side-channels are unavoidable in secure processors? Tramèr et al. [71] 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 [6,51] and later versions of TPM [1], 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 multiparty 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 [6, 22, 27, 31–34, 48, 49, 51, 66, 73]. 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 [1], Arm's Trustzone [5, 7], and Intel's SGX [6, 51]. 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 [27]; others additionally defend against physical snooping of memory buses [33, 34, 49]; 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. [64] 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 [24, 29, 37, 40, 45]. 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 [24], others require sending one or more physical hardware tokens during the protocol [37, 40, 42, 52], thus limiting the protocol's practicality and the hardware token's global reusability. Finally, a couple recent works [42, 52] also adopt the GUC framework to model hardware tokens — however, the use of GUC in these works [42, 52] 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 [24, 37, 45], these works [42, 52] consider the same model where the hardware tokens are passed around between parties during protocol execution, and not realistic secure processors like SGX.

Use of trusted hardware in applications. Numerous works have demonstrated how to apply trusted hardware to design secure cloud systems [11, 28, 50, 61, 62], cryptocurrency systems [74], collaborative data analytics applications [55], and others [12, 23, 59, 63]. 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 [10, 65], however, they either do not support cryptographically sound reasoning [65], or do not support cryptographically sound composition in general protocol design [10].

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 [31, 73]. 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.

3 Formal Definitions, Constructions, and Proofs

In the interest of space, we present our formal definitions, constructions, and proofs in a full version of this work [56] — we refer the reader to the technical roadmap section for an intuitive explanation of the key technical insights, the technicalities that arise in proofs, and how we handle them.

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. Trusted computing group. `http://www.trustedcomputinggroup.org/`.
2. Martín Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. In *Theoretical Aspects of Computer Software*, pages 82–94, 2001.
3. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 20(3):395, 2007.
4. 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.

5. Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3(4):18–24, 2004.
6. Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.
7. ARM Limited. *ARM Security Technology Building a Secure System using TrustZone® Technology*, Apr 2009. Reference no. PRD29-GENC-009492C.
8. 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.
9. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A universally composable cryptographic library. *IACR Cryptology ePrint Archive*, 2003:15, 2003.
10. 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.
11. Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.
12. 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.
13. Florian Bohl and Dominique Unruh. Symbolic universal composability. In *IEEE Computer Security Foundations Symposium*, pages 257–271, 2013.
14. Dan Boneh and Moni Naor. Timed commitments. In *CRYPTO*, 2000.
15. Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS*, 2004.
16. Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing. *IACR Cryptology ePrint Archive*, 2009:95, 2009.
17. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
18. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*. 2007.
19. Ran Canetti and Marc Fischlin. Universally composable commitments. In *Advances in Cryptology (CRYPTO)*, pages 19–40, 2001.
20. Ran Canetti and Jonathan Herzog. Universally composable symbolic security analysis. *J. Cryptology*, 24(1):83–147, 2011.
21. Ran Canetti and Tal Rabin. Universal composition with joint state. In *CRYPTO*, 2003.
22. David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.
23. Chen Chen, Himanshu Raj, Stefan Saroiu, and Alec Wolman. cTPM: A cloud TPM for cross-device trusted applications. In *NSDI*, 2014.
24. Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. Functional encryption from (small) hardware tokens. In *Asiacrypt*, 2013.
25. Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC'86*, pages 364–369, 1986.
26. Victor Costan and Srini Devadas. Intel SGX explained. Manuscript, 2015.
27. Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.
28. 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.

29. 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.
30. Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6), June 1985.
31. 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.
32. Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.
33. 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.
34. 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.
35. Juan Garay, Philip MacKenzie, Manoj Prabhakaran, and Ke Yang. Resource fairness and composability of cryptographic protocols. In *TCC*, 2006.
36. 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.
37. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *CRYPTO*, 2008.
38. Dov Gordon and Jonathan Katz. Complete fairness in multi-party computation without an honest majority. In *TCC*, 2009.
39. 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.
40. Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, 2010.
41. 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.
42. 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.
43. Omer Horvitz and Virgil D. Gligor. Weak key authenticity and the computational completeness of formal encryption. In *CRYPTO*, pages 530–547, 2003.
44. 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.
45. Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, 2007.
46. Bernhard Kauer. Tpm reset attack. `http://www.cs.dartmouth.edu/~pkilab/sparks/`.
47. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

48. 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.
49. 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.
50. 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.
51. 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.
52. 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`.
53. Daniele Micciancio and Bogdan Warinschi. Completeness theorems for the Abadi-Rogaway language of encrypted expressions. *J. Comput. Secur.*, 12(1):99–129, January 2004.
54. Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference (TCC)*, 2004.
55. 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.
56. Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. *IACR Cryptology ePrint Archive*, 2016:1027, 2016.
57. Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *POST*, pages 53–72, 2015.
58. Adam Petcher and Greg Morrisett. A mechanized proof of security for searchable symmetric encryption. In *CSF*, 2015.
59. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.
60. 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.
61. 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.
62. 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.
63. Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *IEEE S&P*, 2005.
64. 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.
65. 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.

66. 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.
67. 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.
68. Mike Szczys. TPM crytography cracked. `http://hackaday.com/2010/02/09/tpm-crytography-cracked/`.
69. 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.
70. Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, August 1984.
71. 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.
72. Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.
73. Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.
74. Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *ACM CCS*, 2016.
75. 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.