ZK + TEE
On realizing NIZK with TEEs

Jan Bobolz[1], Markulf Kohlweiss[1,2], Lorenzo Martinico[1], John Mc. Placeholder

[1] University of Edinburgh, UK
[2] IOG, UK

January 13, 2025

Abstract. Trusted Execution Environments (TEEs) have been used as a mechanism to simplify the implementation of cryptographic protocols by providing what amounts to a verifiably trusted third party. The existence of reliable and trustworthy TEEs can be considered a strong cryptographic assumption. However, like other cryptographic assumptions, formalising protocols that use TEEs in a composable way is not always straightforward, and in particular existing formalisation require adding certain modelling artifacts to the protocols. In this work, we explore how to formulate more realistic protocols by shifting the necessary complexity into the model of the TEE itself rather than the protocol, by focusing on the construction of efficient Non-Interactive Zero Knowledge (NIZK) protocols. In particular, we show how in different settings, a TEE functionality that is both *observable* and *programmable* allows the formulation of more natural protocols than the existing TEE functionality from the literature [PST17].
[**Jan:** Updated list of constructions:
(1) zero-overhead NIZK from $\mathcal{G}$-$\texttt{Att}^{\text{obsprog}}$.
(2) non-succinct NIZK from crs + $\mathcal{G}$-$\texttt{Att}^{\text{vanilla}}$.
(3) potentially a succinct Fischlin-style NIZK with GROM and $\mathcal{G}$-$\texttt{Att}^{\text{vanilla}}$.
(4) A client/server version, where prover client doesn't need a TEE.
(5) Maybe a "combiner" version, which is secure if the TEE is secure or Groth16 is secure.]
Additionally, we show that a TEE with certain vulnerabilities is still sufficient to provide a secure NIZK protocol. We investigate how the introduction of rollback attacks, partial leakage attacks, and attestation forgeries affect the security of the protocol [**Lorenzo:** Probably quite optimistic; maybe threshold protocols?] [**Jan:** Current thoughts:
(6) Security still holds in presence of rollback attacks (because stateless TEE; maybe result not in *this* paper)
(7) Some versions are secure in transparent setting (sealed-glass style, or Pooya style where even honest parties' randomness (and input?) is leaked to adversary).] [**Markulf:** (8) non-anonymous attestation prove servers might make for a nice story, as they are not supported by Pass, neither in terms of functionality, nor proof strategy.]

lorenzo.martinico@ed.ac.uk

Table of Contents

1 Introduction

We introduce two protocols that UC-realise NIZKs using TEEs, with different setup assumptions.

1.1 Structure of the paper

In section **??**, we introduce the $\mathcal{G}$-`Att`$^{\text{vanilla}}$ functionality from [PST17], and in section **??**, we introduce a simple protocol Π^V to realise fnizk in the $\mathcal{G}$-`Att`$^{\text{vanilla}}$-hybrid world, where the prover has access to TEEs (inspired by [TZL+16]). The protocol requires the introduction of a backdoor [**Lorenzo:** explain] to allow the simulator to extract the witness for honest parties.

We then propose, in section **??**, $\mathcal{G}$-`Att`$^{\text{obsprog}}$, an extension to $\mathcal{G}$-`Att`$^{\text{vanilla}}$ that adds programmability and observability, inspired by []. In section **??** we show how to construct Π^0, a simpler protocol than Π^V by replacing $\mathcal{G}$-`Att` with $\mathcal{G}$-`Att`$^{\text{obsprog}}$. We then show how to adapt Π^0 to the case where the proof of knowledge can be outsourced to an untrusted remote party with access to a TEE (protocol Π^Ω).

In section **??**, we discuss how to augment the previous protocols to a setting where the security of the TEEs is degraded...

1.2 Related work

collect related results: sealed glass, Bingshengs paper, Microsoft article,

FHE TEE paper, running FHE inside a TEE. Useful for transparency

2 Preliminaries

—add notation for party specific initialization code for protocols.

2.1 Notation

2.2 Pseudocode notation and UC

[**Jan:** I think what is below right now is probably more correct than not, but feels like crap, didactically and story-wise. Probably not something we'd include in this paper.] [**Markulf:** I think it's fine for the appendix? We can advertise and sell it a bit in the body.]

Consequences of this notation (compared to "full Canetti"):

- In contrast to Canetti, in our notation it cannot be that two ITMs are subroutines *of each other*. The session ID rules prohibit the "deeper" ITMs from sending messages to a shallower ITM (in that direction, information only flows via return values). As an example, a channel would never activate/send a message to a protocol machine. Protocols would have to poll the channel. The advantage is more natural pseudocode: When writing pseudocode in Canetti UC, strictly speaking you have to write a handler for messages that the channel passes to the protocol, which then parses the message and picks the right "next steps" to execute. That works okay in English, because it's imprecise enough to get around those details (at the detriment of rigor). It also means that protocols need to be written in an "explicitly save state to memory, so that when the next protocol message arrives, we can retrieve the state and continue"-style.

  In our notation, the protocol can be just written down step-by-step within a single procedure, with no need to explicitly save state between rounds. Whenever the protocol expects to receive a network message, it would simply call $m \leftarrow \mathsf{Ch} :: \mathcal{P}.\textsc{Receive}()$ and as soon as a message arrives, it would simply continue its execution (with the previous state still there and with the received m).

[**Lorenzo:** this Functionality is still hiding things like who are the parties and subroutines allowed to be interacted / called from the protocol. If we are in a static setting, would it make sense to explicitly define variables for the set of parties and a call graph for which party is allowed to talk to which Procedure? i.e. P.Prove(x) for P in set Provers] [**Markulf:** Yes, we have used the notation $P \in Parties$ in the past, it is fine. I don't think we need a callgraph to determin which party can call which procedure, procedures are party respecting, $P.$ procedures can only call $P.$ procedures.]

Functionality 1: Notation for UC protocols/functionalities

$\underline{\text{INIT}_{sid}()}$
// Executed when ITM comes into existence
// Variables initialized here are accessible to all procedures of this ITM
// sid is the session ID of this ITM [**Jan:** Where does it come from?]

$\underline{\mathcal{P}.\text{PROC}_{sid}(x)}$ // Behavior when party $\mathcal{P}$ invokes procedure PROC with (procedure) session sid and input x
// [...]
assert $\phi(x)$ // Equivalent to "If not $\phi(x)$, then return $\perp$"
// [...]
$y' \leftarrow \mathcal{F} :: \mathcal{P}.\text{PROC}'_{sid'}(x')$ // Calls (passes control to) protocol/functionality $\mathcal{F}$. Resumes execution after $\mathcal{F}$ returns.
// [...]
return y // Output the result of this procedure to its caller.

– There is no dynamic spawning of ITMs with dynamic code. If a procedure is not specified at design time, it can never be called.

Notes[**Lorenzo:** should this be before the previous itemize? you reference the sid rules there but they are defined here]

– When a procedure with name $\mathcal{P}.\text{PROC}_{sid}$ is invoked, it can only invoke other procedures with the same $\mathcal{P}$ in their names. The $\mathcal{P}$ in the procedure name doesn't have to be the same as any ITI's party ID (e.g., an ideal functionality provides procedures for many different parties).[**Lorenzo:** confused by what you mean with this; if parties are static, aren't we assuming that we have n copies of Proc, one for each party that has access to that subroutine in the functionality? what is $\mathcal{P}$ if not the PID?] [**Markulf:** I am also confused, do we even have ITI's, they would only appear when we give a mapping from SSP-UC to Canetti UC.]

– There is a hierarchical structure to sessions. A procedure $\mathcal{P}.\text{PROC}_{sid}$ can only invoke procedures $\mathcal{P}.\text{PROC}'_{sid'}$ where sid is a prefix of sid'. The sid in the procedure name doesn't have to be the same as any ITI's session ID (e.g., a global functionality provides procedures for many different sessions).

2.3 Functionalities

(Functionality 3 from G-GG paper)

Discuss tradeoffs and variants of functionality: - mauling - anonymity - restrict prover set (necessary for TEE-enabled provers)

2.4 Modelling Trusted Hardware in UC

[PST17] provides the first formulation of TEE as a G(eneralised)UC functionality. Their $\mathcal{G}$-`Att` global functionality allows minimal state sharing across different sessions; in particular, the public key to the attestation signature scheme is shared across all protocol sessions that use $\mathcal{G}$-`Att`. All other functionality operations are session-bounded by including the session id as part of the enclave attestation signature. The adversary is allowed to install an enclave under a different session

Functionality 2: Interpretation of our notation in Canetti's UC [Can20]

[**Jan:** I'm citing the ePrint as "Can20" instead of "Can00". Does anyone see an issue with this?] [**Markulf:** No thats fine, your comment can be closed. Maube discuss UC history in preliminaries.]
Let sid^* be this ITM's session ID. Let $\mathcal{P}^*$ be this ITM's party ID.
Assumption for this description: For any $\mathcal{P}, sid$, there is at most one ITI in the system with that party ID and session ID. [**Jan:** For sanity, to address a message to a ITI without having to go into weird details with its code. Looks like the assumption is not necessarily true in Canettiland: "Furthermore, the model allows multiple ITIs in an execution to have the same (non-extended) identity - as long as they have different programs."][**Lorenzo:** is the code of our ITMs the union of all $\mathcal{P}.proc$ for all protocols/functionalities defined in the experiment for the current session?][**Jan:** The code of the ITMs is the code listed below. The code listed below interprets the code we write in SSP-UC notation. Essentially, each box corresponds to an ITM and the procedures defined in that box are the ones available for interpretation in the code below.]

Upon first activation

1: Initialize a table $T[\cdot] \leftarrow \bot$ to store procedure state.
2: Initialize a table $G \leftarrow ()$ to store "ITM-local" state (shared between procedures and procedure invocations).
3: Run the code specified in INIT_{sid^*}.
4: Proceed below to process the message that initially activated this ITM. [**Lorenzo:** can i initialise without sending another Proc message?] [**Jan:** Initialization just always happens when the ITI is first activated. So I'd say you'd never invoke INIT explicitly. If you want to create a signature, you just send $(h_{\text{caller}}, \text{SIGN}, sid, m)$ to $\mathcal{F}$-Sig, which (if this message spawns it into existence) will secretly run INIT first and then SIGN.]

Upon receiving input or backdoor input $(h_{\text{caller}}, \text{PROC}, sid, x)$ from a machine with party ID $\mathcal{P}_{\text{caller}}$ and session ID sid_{caller}

5: If no procedure with name $\mathcal{P}_{\text{caller}}.\text{PROC}_{sid}$ exists, then return $\bot$.
6: If sid_{caller} is not the longest substring of sid [**Jan:** How to decide? SSP-UC: Static check! Simplified alt: sid is exactly one segment longer than sid'], then return $\bot$.
7: Choose a new random (unguessable) handle h for this procedure call.
8: [**Lorenzo:** should $\mathcal{P}$ be $\mathcal{P}^*$] [**Jan:** $\mathcal{P}^*$ is never used in this framework. But I changed the undefined $\mathcal{P}$ to $\mathcal{P}_{\text{caller}}$.]
9: $T[h] \leftarrow (\mathcal{P}_{\text{caller}}, sid_{\text{caller}}, h_{\text{caller}}, ctr = 0, \text{PROC}, sid, x)$ // Initialize procedure state
10: Execute the instructions specified in $\mathcal{P}_{\text{caller}}.\text{PROC}_{sid}$. [**Jan:** Remove if unused:] If $\mathcal{P}_{\text{caller}}$ is corrupted and a procedure $\mathcal{P}^{\text{corrupt}}_{\text{caller}}.\text{PROC}_{sid}$ is explicitly specified, execute that instead. [**Lorenzo:** how do we know a party is corrupted? is there a directory ITI? is corruption per-session] [**Jan:** Corruption is per-party (if party has multiple sessions, all of them are corrupted). I don't 100% know how Canetti knows which parties are corrupted. In our SSP-UC mind, when you corrupt a party, it's added to a globally readable set $\mathcal{C}$ and that's that.]
11: After each instruction, move the program counter ctr to keep track of the current instruction (usually, ctr is incremented to point to the next instruction; in case of branching, ctr may be set to point to a specific instruction).
12: Instructions that read/write local variables' values read/write them from/to $T[h]$ (except that $\mathcal{P}_{\text{caller}}, sid_{\text{caller}}, h_{\text{caller}}, ctr$ are inaccessible and $\mathcal{P}_{\text{caller}}, \text{PROC}, sid$ are readonly).[**Markulf:** $\mathcal{P}_{\text{caller}}$ appeared twice, inaccessible or readonly?] Instructions that create new local variables simply append the variable name/value to $T[h]$.[**Lorenzo:** T and G seem to simulate the tapes of an ITM; are our ITMs (ITIs?) stateless i.e. just code no data?] [**Jan:** The SSP-UC notation (which we're translating here to Canetti) doesn't talk about ITMs at all. In that notation, each procedure has state and each "box" (∼ITI) has state accessible to all its procedures (it's like a class in OOP). T/G model that procedure/box state as Cantti ITM state. So in short: T and G only exist to write the inherent state of boxes and procedures in the SSP-UC notation as an artifact that an ITM/ITI can store in Canettiland.]
13: Instructions that read ITM-local variables' values read them from G. Instructions that create new ITM-local variables simply append the variable name/value to G.
14: Upon encountering an external call instruction "$y' \leftarrow \mathcal{F} :: \mathcal{P}.\text{PROC}'_{sid'}(x')$", do the following:
15: Send $(h, \text{PROC}', sid', x')$ to (the subroutine input tape of) the appropriate ITI (there is at most one ITM in the system that exposes $\mathcal{P}.\text{PROC}'_{sid'}$). [**Jan:** Weaseling out of specifying the recipient.]
16: // This yields activation to $\mathcal{F}$. Execution resumes below when receiving subroutine ouput (h, y')
17: Upon encountering an adversary call instruction "$y' \leftarrow \mathcal{A} :: \mathcal{P}.\text{PROC}'_{sid^*}(x')$", do the following:
18: Send $(h, \text{PROC}', sid^*, x')$ to (the backdoor tape of) the adversary $\mathcal{A}$.
19: // This yields activation to $\mathcal{A}$. Execution resumes below when receiving backdoor tape message (h, y')
20: Upon encountering an instruction "**return** y", do the following:
21: Set $T[h] \leftarrow \bot$. // Clear procedure state.
22: Send (h_{caller}, x') to (the subroutine output tape of) $(\mathcal{P}_{\text{caller}}, sid_{\text{caller}})$.
23: // This yields activation to the caller.

Upon receiving subroutine output or backdoor tape message (h, y')

24: If $T[h] = \bot$, ignore the message.
25: Otherwise, retrieve procedure state $(\mathcal{P}_{\text{caller}}, sid_{\text{caller}}, h_{\text{caller}}, ctr = 0, \text{PROC}, sid, x, \dots) \leftarrow T[h]$. [**Markulf:** Sanity check, it shouldn't be $ctr = 0$?]
26: The counter ctr points to an instruction $z \leftarrow \mathcal{P}_{\text{caller}}.\text{PROC}'_{sid}(x')$.
27: Set $z = y'$ within $T[h]$ and increment ctr within $T[h]$. Go to Line 10.

Upon receiving backdoor tape message corrupt

28: Send $(\text{corrupt}, T)$ to the adversary's backdoor tape. [**Jan:** plus the global state. Give it a name.] [**Markulf:** Record which party gets corrupted? For ideal functionalities we don't necessarily reveal the state?]

Functionality 3: $\mathcal{F}$-NIZK$[\mathcal{R}, \mathbf{P}, \mathrm{V}]$

$\underline{\textsc{Init}_{sid}()}$
1: $T \leftarrow [\,]$

$\underline{\mathcal{P}.\textsc{Prove}_{sid}(x, w)}$ for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
2: **if** $(x, w) \notin \mathcal{R}$ **then return** $\perp$
3: $\pi \leftarrow$ `Adv` :: $\mathcal{P}.\textsc{Simulate}_{sid}(x)$
4: $T \leftarrow T \cup (x, \pi)$
5: **return** π

$\underline{\mathcal{P}.\textsc{Verify}_{sid}(x, \pi)}$ for $\mathcal{P} \in \mathrm{V} \setminus \mathbf{C}$
6: **if** $(x, \pi) \in T$ **then return** 1
7: $w \leftarrow$ `Adv` :: $\mathcal{P}.\textsc{Extract}_{sid}(x, \pi)$
8: **if** $(x, w) \in \mathcal{R}$ **then** $T \leftarrow T \cup (x, \pi)$
9: **if** $(w = \text{maul} \wedge (x, *) \in T)$ **then** $T \leftarrow T \cup (x, \pi)$
10: **if** $(x, \pi) \in T$ **then**
11: **return** 1
12: **else**
13: **return** 0

Purple changes on line 3 and 7 make the NIZK functionality non-anonymous, the purple changes on line 9, make the functionality weak, i.e. allow for mauling of proofs.

TODO: Describe implications of GUC and session in particular wrt signature verification
Transparent enclave, semi-transparent, rollback enclave; AGATE unifies them

Functionality 4: $\mathcal{G}$-Att$^{\text{vanilla}}[\lambda, \mathsf{reg}, \Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})]$ [PST17]

$\underline{\textsc{Init}_{sid}}$
1: $(pk_{\text{att}}, sk_{\text{att}}) \leftarrow \mathsf{Gen}(1^\lambda)$
2: $\mathsf{Encl}[\cdot, \cdot] \leftarrow \perp$

$\underline{\mathcal{P}.\textsc{getpk}_{sid}()}$ for all $\mathcal{P}, sid$
3: **return** pk_{att}

$\underline{\mathcal{P}.\textsc{install}_{sid}(idx, \mathsf{prog})}$ for $\mathcal{P} \in \mathsf{reg}$ and all sid
4: $eid \xleftarrow{\$} \{0,1\}^\lambda$
5: **if** $\mathcal{P}$ is not corrupted **then**
6: $idx \leftarrow sid$ [**Markulf:** could we use sid' instead of idx? Or did for declared (session) id? Could also abort if they differ for honest parties? Or provide a more powerful $\mathcal{P}^{\text{corrupt}}.\textsc{install}$. Also what would go wrong if we removed this line?][**Jan:** Options: all feasible, but let's not stray too much from Pass's writeup. Re "What would go wrong?": Environment could spin up an enclave for the target session and compute a proof without our simulator seeing any of it.]
7: $\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st = ())$
8: **return** eid

$\underline{\mathcal{P}.\textsc{resume}_{sid}(eid, input)}$ for $\mathcal{P} \in \mathsf{reg}$ and all sid
9: **assert** $\mathsf{Encl}[eid, \mathcal{P}] \neq \perp$
10: $(\mathsf{prog}, idx, st) \leftarrow \mathsf{Encl}[eid, \mathcal{P}]$
11: $(output, st') \leftarrow \mathsf{prog}(input, st)$ [**Jan:** explicit randomness?]
12: $\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st')$
13: $meas \leftarrow (idx, eid, \mathsf{prog}, output)$
14: $\sigma \leftarrow \mathsf{Sign}(sk_{\text{att}}, meas)$
15: **return** $(output, \sigma)$

3 passgatt

We provide a protocol to implement NIZKs[R] using the passgatt functionality. The protocol UC emulates the weak NIZK functionality of [BFKT24] (functionality 3).

The protocol proceeds between Prover and Verifier.

Protocol 1: $\Pi^{V}[\mathbf{P}, \mathrm{V}]$

$\mathcal{P}.\textsc{Prove}(x, w)$ for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$

1: $crs \leftarrow \mathcal{F}\text{-}\texttt{CRS} :: \mathcal{P}.\textsc{crs}()$
2: $eid \leftarrow \mathcal{G}\text{-}\texttt{Att}^{\text{vanilla}} :: \mathcal{P}.\textsc{install}(\mathcal{P}.sid, \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]})$
3: $((x, crs), \sigma_\pi) \leftarrow \mathcal{G}\text{-}\texttt{Att}^{\text{vanilla}} :: \mathcal{P}.\textsc{resume}(eid, (\mathsf{Prove}, crs, x, w))$
4: **return** (eid, σ_π)

$\mathcal{V}.\textsc{Verify}(x, \pi)$ for $\mathcal{V} \in \mathrm{V} \setminus \mathbf{C}$[**Lorenzo:** is this correct? verifiers should not just be provers?][**Jan:** Hotfixed notation, but want to get rid of any (non-session) restrictions on verifiers long-term.]

5: $(eid, \sigma) \leftarrow \pi$
6: $crs \leftarrow \mathcal{F}\text{-}\texttt{CRS} :: \mathcal{V}.\textsc{crs}()$
7: $pk_{\text{att}} \leftarrow \mathcal{G}\text{-}\texttt{Att}^{\text{vanilla}} :: \mathcal{V}.\textsc{getpk}()$
8: $b \leftarrow \Sigma.\mathsf{Ver}(pk_{\text{att}}, (\mathcal{V}.sid, (sid, eid, \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}, (x, crs))))$
9: **return** b

Program 2: $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$

$\mathsf{Prove}(crs, x, w)$

1: **assert** $(x, w) \in \mathcal{R}$[**Lorenzo:** gatt semantics: if assert failed, return bot with or without attestation?]
2: **return** (x, crs)

$\mathsf{Simulate}(\tau, x)$

3: $crs \leftarrow \mathsf{CRS}(\tau)$
4: **return** (x, crs)

3.1 Simulator

Simulator 1: `Sim` for Protocol 1

$\mathcal{P}.\text{INSTALL}_{sid}(idx, \mathsf{prog}_{\text{ZK}[\mathcal{R}]})$ from corrupted party $\mathcal{P}$ to $\mathcal{G}$-`Att`$^{\text{vanilla}}$

1: **if** $\bar{\mathcal{P}} = \perp$ **then**
2: $\quad \bar{\mathcal{P}} \leftarrow \mathcal{P}$ [**Jan:** What if this isn't set before we read it in SIMULATE?][**Lorenzo:** If we know the set of corrupted parties in reg we could pick one randomly?]
3: $eid \leftarrow \mathcal{G}$-`Att`$^{\text{vanilla}} :: \mathcal{P}.\text{INSTALL}_{sid}(idx, \mathsf{prog}_{\text{ZK}[\mathcal{R}]})$ [**Lorenzo:** what do we do if idx neq sid?][**Jan:** No action needed. Let adv do whatever it wants. Potentially cannot happen at all, depending on view on UC]
4: **return** eid

$\mathcal{P}.\text{RESUME}_{sid}(eid, (\mathsf{Prove}, (crs, x, w)))$ from corrupted party $\mathcal{P}$ to $\mathcal{G}$-`Att`$^{\text{vanilla}}$

5: **if** $(x, w) \in \mathcal{R}$ **then** [**Lorenzo:** do we need to check signature is valid?] [**Jan:** No, we only care about gathering *at least* all the witnesses that corrupted parties submit to enclaves. It's fine/easier to overestimate. I've simplified this a bit.]
6: $\quad E[x] \leftarrow w$
7: **return** $\mathcal{G}$-`Att`$^{\text{vanilla}} :: \mathcal{P}.\text{RESUME}(eid, (\mathsf{Prove}, (crs, x, w)))$

$\text{SIMULATE}_{sid}(x)$ [**Lorenzo:** do we need different code for the non-anonymous case?][**Jan:** Yes, but do we even want to model that case?]
8: $(crs, \tau) \leftarrow \mathcal{F}$-`CRS-Sim` $:: \text{TRAPDOOR}()$[**Lorenzo:** do we need to state more formally that the crs distribution is the same as in the real world?]
9: $eid \leftarrow \mathcal{G}$-`Att`$^{\text{vanilla}} :: \bar{\mathcal{P}}.\text{INSTALL}_{sid}(sid, \mathsf{prog}_{\text{ZK}[\mathcal{R}]})$
10: $((x, crs), \sigma) \leftarrow \mathcal{G}$-`Att`$^{\text{vanilla}} :: \bar{\mathcal{P}}.\text{RESUME}_{sid}(eid, (\mathsf{Simulate}, (\tau, x)))$
11: **return** (eid, σ)

$\text{EXTRACT}_{sid}(x, \pi)$
12: $(crs, \tau) \leftarrow \mathcal{F}$-`CRS-Sim` $:: \text{TRAPDOOR}()$
13: parse $(eid_\pi, \sigma_\pi) \leftarrow \pi$
14: assert $\Sigma.\mathsf{Ver}(pk_{\text{att}}, (\sigma_\pi, (idx, eid_\pi, \mathsf{prog}_{\text{ZK}[\mathcal{R}]}, (x, crs))))$ // Returns invalid witness $\perp$ if π is invalid.
15: [**Lorenzo:** what is the correct behaviour for a simluator if extract contains invalid proof but simulator has seen a valid witness for x? what if $\pi = Proof(x, w')$ and both $(x, w) \in R \wedge (x, w') \in R$?][**Lorenzo:** what if we have not observed prove(x,w)?] [**Jan:** Task of the simulator is to output a witness or maul iff the proof is valid. This is to be indistinguishable from the real world, where Verify outputs 1 iff the proof is valid. If multiple witnesses make sense, F-NIZK makes no restrictions as to which witness the simulator shall return. Any witness will do.]
16: $w \leftarrow E[x]$
17: **if** $(x, w) \notin \mathcal{R}$ **then** // Equivalent to $w = \perp$? Or replicate set T? Seems unnecessary.
18: $\quad$ **return** maul // Relevant for weak-EUF-CMA signature schemes Σ. Cannot happen for strong EUF-CMA (except if forgery).
19: **return** w

We construct Simulator 1 to reproduce the behaviour of Protocol 1. The simulator produces proofs in the ideal world that correspond to the format of real world proofs i.e. attestation signatures over the output of program $\mathsf{prog}_{\text{ZK}[\mathcal{R}]}$. Rather than using the honest Prove interface to the enclave, the simulator can use its access to trapdoor τ to produce a valid signature without having access to the witness through the $\mathsf{Simulate}$ interface. When the environment generates a proof for a corrupted prover by running the $\mathsf{prog}_{\text{ZK}[\mathcal{R}]}$ enclave, the simulator records its inputs. For any subsequent call from the $\mathcal{F}$-`NIZK` functionality to extract from a valid proof, the simulator can return the witness if it has observed the corresponding enclave call previously, or the special message "maul" if the witness is not valid.

[**Lorenzo:** If sigma is weaker than Pass (modgatt?), it is possible to provide signatures that attest to some F(meas) ->? mauling?]

We now argue that the behaviour of protocol Π^V interacting with an arbitrary adversary is indistinguishable to the ideal functionality interacting with the simulator.

Cases:

1) adversary can create a proof that verifies for x where it doesn't have a valid witness

2) break ZK

3) extraction

4) do something bad with sessions

5) mauling behaves the same

4 programmable gatt

[**Jan:** Passgatt paper says:
The environment Z cannot install an enclave with the challenge sid without going through A [which is not true, imo, Z should be able to make corrupted calls from other sessions];
The environment Z cannot access any enclave installed by a corrupt party without going through A.
That's weird because we're now doing observation for exactly the case that the environment does something in another session that may impact our session. But that was simply *not possible* before.] We now describe a new version of modgatt which allows adversarial programming for the output of existing enclaves in a way that is detectable in the real world but not in the ideal world. This is inspired by the setting of global restricted programmable random oracles [CDG+18].

Given a program that securely implements a NIZK prover in a similar manner to **??**, we want to enable the simulator to perform proof simulation and witness extraction without the need to provide a CRS trapdoor.

We augment the modgatt functionality with two new interfaces: Program() and IsProgrammed(). The Program interface allows a corrupted party to provide attestation signatures on arbitrarily chosen enclave outputs. IsProgrammed() returns a boolean value depending on whether an enclave output has been attested through the programming interface, and can be called in conjunction with Verify() to evaluate a trust assumption with a remote enclave. Assuming that the enclave output includes the claimed session for the enclave (standard in Passgatt, depends on $\mathbb{S}$ in AGATE), modgatt will only respond to IsProgrammed only if the caller's session id is the same as the program output.

[**Jan:** Potentially interesting sanity-check: observable programmable GROM from observable programmable Gatt (observed by Pooya). Caveat: censorable GROM] [**Jan:** Interesting note: It's important that the attestation signature is a group signature b/c NIZK should not leak identity of prover.]

A programmable gatt allows us to realize fnizk through a protocol that contains a more natural enclave program for letting the server compute the enclave. We define $\bar{ZK}[R]$ to be a program with interface {Init,Prove}. The code of these subroutines of $\bar{ZK}[R]$ and the corresponding protocol are the same as in **??**, with all references to the CRS removed (redefine here?) and protocol **??** installing $\bar{ZK}[R]$ instead of $ZK[R]$.

The simulation strategy involves running the $\bar{ZK}[R]$ in the simulator's head and producing valid proofs through attestation programming, while hiding this from the corrupted parties.

More precisely, the simulator for the ideal functionality (under server corruption) proceeds as follows: on initialisation, it instructs modgatt to install program $\bar{ZK}[R]$ on behalf of the server. Rather than running the Init subroutine in this enclave, it produces a public encryption key pk locally, and calls $Program((sid, eid, \bar{ZK}[R], pk))$. It returns eid and the resulting sigma to the environment.

On any subsequent prove call (to the server) with some ciphertext ct, it decrypts the ciphertext to (x, w), stores those values and calls FNIZK.prove(x,w). When queried with proof simulation, it responds with ct and the result value for $Program((sid, eid, \bar{ZK}[R], (ct, x)))$

On an extract call from the ideal functionality for (x, π), the simulator parses $\pi = (ct, \sigma)$, decrypts ct to x, w INSERT MAULING STUFF? and returns w.

If the server (or any malicious party in the session) attempts to verify any of the attestation signatures produced by the enclave by calling IsProgrammed, the simulator suppresses the call and always responds with false.

Functionality 5: $\mathcal{G}$-Att$^{\text{obsprog}}[\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver}), \mathsf{reg}]$

$\underline{\text{INIT}_{sid}}$

1: // $(pk_{\text{att}}, \mathsf{Sign}) \leftarrow$ Adv :: INITIALIZE() [**Jan:** Okay to get rid of Adv here? To avoid discussion about "which simulator/adversary" we mean?]
2: $(pk_{\text{att}}, sk_{\text{att}}) \leftarrow \mathsf{Gen}(1^\lambda)$
3: $\mathsf{Sig}[\cdot] \leftarrow \bot$
4: $\mathsf{Encl}[\cdot, \cdot] \leftarrow \bot$
5: Obs $\leftarrow$ []
6: Prog $\leftarrow \varnothing$

$\underline{\mathcal{P}.\text{GETPK}_{sid}()}$ for all $\mathcal{P}, sid$

7: **return** pk_{att}

$\underline{\mathcal{P}.\text{VERIFY}_{sid}(\sigma, meas)}$ for all $\mathcal{P}, sid$

8: **return** $meas \overset{?}{\in} \mathsf{Sig}[\sigma]$

$\underline{\mathcal{P}.\text{OBSERVE}_{sid}()}$ for all $\mathcal{P}, sid$

9: **return** Obs

$\underline{\mathcal{P}.\text{PROGRAM}_{sid}(idx, eid, \mathsf{prog}, output)}$ for all $\mathcal{P}, sid$

10: **assert** $sid = idx$
11: $meas \leftarrow (idx, eid, \mathsf{prog}, output)$
12: $\sigma \leftarrow \mathsf{Sign}_{sk_{\text{att}}}(meas)$
13: $\mathsf{Sig}[\sigma] \leftarrow \mathsf{Sig}[\sigma] \cup \{meas\}$
14: $\mathsf{Prog} \leftarrow \mathsf{Prog} \cup \{meas\}$
15: **return** σ

[**Jan:** Capitalization of GETPK etc.]

$\underline{\mathcal{P}.\text{INSTALL}_{sid}(idx, \mathsf{prog})}$ for $\mathcal{P} \in \mathsf{reg}$ and all sid

16: $eid \overset{\$}{\leftarrow} \{0,1\}^\lambda$
17: **if** $sid \neq idx$ **then** // caller session $\neq$ enclave session
18: Obs $\leftarrow$ Obs : $(\mathcal{P}, \text{INSTALL}, idx, eid, \mathsf{prog})$
19: [**Jan:** previously forced $idx = sid$ if honest] [**Markulf:** Add this back before line 17 to prevent honest users from ever being observed?] [**Jan:** Using wrong idx is a programming mistake in practice, should allow env to make it (security in presence of buggy software). Why doesn't GROM disallow querying (sid', x) for honest users? This differentiates us from Pass]
20: $\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st = ())$
21: **return** eid

$\underline{\mathcal{P}.\text{RESUME}_{sid}(eid, input)}$ for $\mathcal{P} \in \mathsf{reg}$ and all sid

22: **assert** $\mathsf{Encl}[eid, \mathcal{P}] \neq \bot$
23: $(\mathsf{prog}, idx, st) \leftarrow \mathsf{Encl}[eid, \mathcal{P}]$
24: $(output, st') \leftarrow \mathsf{prog}(input, st)$
25: $\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st')$
26: $meas \leftarrow (idx, eid, \mathsf{prog}, output)$
27: $\sigma \leftarrow \mathsf{Sign}_{sk_{\text{att}}}(meas)$
28: $\mathsf{Sig}[\sigma] \leftarrow \mathsf{Sig}[\sigma] \cup \{meas\}$
29: **if** $sid \neq idx$ **then** // caller session $\neq$ enclave session
30: Obs $\leftarrow$ Obs : $(\mathcal{P}, \text{RESUME}, input, meas)$
31: **return** $(output, \sigma)$

$\underline{\mathcal{P}.\text{ISPROGRAMMED}_{sid}(idx, eid, \mathsf{prog}, output)}$ for all $\mathcal{P}, sid$

32: **assert** $sid = idx$
33: **return** $(idx, eid, \mathsf{prog}, output) \overset{?}{\in} \mathsf{Prog}$

[**Markulf:** To be merged with VERIFY][**Jan:** Any session is able to verify, but IsProgrammed is session-gated. Don't merge, I think.]

5 0-overheads

Define the "natural" Protocol between prover and verifier in the presence of Functionality 5

Protocol 3: π

$\underline{\mathcal{P}.\text{PROVE}_{sid}(x, w)}$ for $\mathcal{P} \in \mathcal{G}$-Att$^{\text{obsprog}}$.reg

1: $eid \leftarrow \mathcal{G}$-Att$^{\text{obsprog}}$ :: $\text{INSTALL}_{sid}(sid, \mathsf{prog}_{\text{ZK}[\mathcal{R}]})$
2: $(x, \sigma) \leftarrow \mathcal{G}$-Att$^{\text{obsprog}}$ :: $\text{RESUME}_{sid}(eid, (\mathsf{prove}, x, w))$
3: **return** (eid, σ)

$\underline{\mathcal{V}.\text{VERIFY}_{sid}(x, \pi)}$

4: parse $(eid, \sigma) \leftarrow \pi$
5: **return** $\mathcal{G}$-Att$^{\text{obsprog}}$ :: $\text{VERIFY}_{sid}(\sigma, (sid, eid, \mathsf{prog}_{\text{ZK}[\mathcal{R}]}, x))$
6: $\wedge \neg \mathcal{G}$-Att$^{\text{obsprog}}$ :: $\text{ISPROGRAMMED}_{sid}(sid, eid, \mathsf{prog}_{\text{ZK}[\mathcal{R}]}, (sid, eid, \mathsf{prog}_{\text{ZK}[\mathcal{R}]}, x))$

[**Jan:** Hm. In this construction, we're restricting the prover set to just the people who have TEEs. So we're not actually implementing $\mathcal{F}$-NIZK right now, since proving fails for non-TEE-havers. Probably should restrict $\mathcal{F}$-NIZK to set of preapproved provers?!] [**Lorenzo:** Yes, this is good motivation for why we want the 3rd-party server, it allows to unrestrict the set of provers]

Program 4: $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$

$\mathsf{Prove}(x, w)$

1: **assert** $(x, w) \in \mathcal{R}$ [**Lorenzo:** gatt semantics: if assert failed, return bot with or without attestation?]
2: **return** x

Theorem 1. *Protocol 3 UC-emulates* $\mathcal{F}$-NIZK$[\mathcal{R}, \mathsf{reg}]$ *in the presence of* $\mathcal{G}$-Att$^{\mathrm{obsprog}}[\Sigma, \mathsf{reg}]$ *for any EUF-CMA signature scheme* Σ. [**Jan:** *Weak or strong EUF-CMA? This will correspond to weak/strong sim-ext in F-NIZK (purple maul line or not). Also: need definition of (s)EUF-CMA.*] [**Jan:** *Removed existence requirement for corrupted party. Felt unnatural. Anyone can program* $\mathcal{G}$-Att$^{\mathrm{obsprog}}$ *now.*]

Proof: We construct a simulator Adv for the UC emulation experiment. Adv runs a copy of adversary $\mathcal{A}$ internally.

Simulator 2: Adv for Protocol 3

$\textsc{Init}_{sid}$

1: $\mathsf{Prog}_{\mathcal{A}} \leftarrow \varnothing$ // Observed Program queries by $\mathcal{A}$
2: $\Pi = \varnothing$ // Observed $\textsc{Resume}(\cdot, x, w)$ queries by $\mathcal{A}$

$\mathcal{A} :: \mathcal{P}.\textsc{Program}_{sid}(idx, eid, \mathsf{prog}, output)$ for corrupted $\mathcal{P}$ // Dummy $\mathcal{A}$ instructed by environment

1: **assert** $sid = idx$
2: $\mathsf{Prog}_{\mathcal{A}} \leftarrow \mathsf{Prog}_{\mathcal{A}} \cup \{(idx, eid, \mathsf{prog}, output)\}$
3: **return** $\sigma \leftarrow \mathcal{G}$-Att$^{\mathrm{obsprog}} :: \mathcal{P}.\textsc{Program}_{sid}(idx, eid, \mathsf{prog}, output)$

$\mathcal{A} :: \mathcal{P}.\textsc{IsProgrammed}_{sid}(idx, eid, \mathsf{prog}, output)$ for corrupted $\mathcal{P}$ // Dummy $\mathcal{A}$ instructed by environment

3: **assert** $sid = idx$
4: **return** $(idx, eid, \mathsf{prog}, output) \stackrel{?}{\in} \mathsf{Prog}_{\mathcal{A}}$ // Excludes programming done by Adv during Simulate

$\mathcal{A} :: \mathcal{P}.\textsc{Resume}_{sid}(eid, \mathsf{Prove}, (x, w))$ for corrupted $\mathcal{P}$ // Dummy $\mathcal{A}$ instructed by environment

5: **if** $(x, w) \in \mathcal{R}$ **then**
6: $\quad \Pi \leftarrow \Pi \cup \{(x, w)\}$
7: **return** $\mathcal{G}$-Att$^{\mathrm{obsprog}} :: \mathcal{P}.\textsc{Resume}_{sid}(eid, \mathsf{Prove}, (x, w))$

$\mathcal{P}.\textsc{Simulate}_{sid}(x)$ for $\mathcal{P} \in \mathsf{reg}$ // $\mathcal{F}$-NIZK asks for simulated proof [**Lorenzo:** what if $\mathcal{P}$ is honest here? use proxy?]

8: $eid_{\mathcal{P}} \leftarrow \mathcal{G}$-Att$^{\mathrm{obsprog}} :: \mathcal{P}.\textsc{Install}_{sid}(sid, \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]})$
9: $\sigma_\pi \leftarrow \mathcal{G}$-Att$^{\mathrm{obsprog}} :: \mathcal{P}.\textsc{Program}_{sid}(sid, eid_{\mathcal{P}}, \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}, x)$
10: **return** $\pi = (eid_{\mathcal{P}}, \sigma_\pi)$

$\mathcal{P}.\textsc{Extract}_{sid}(x, \pi)$ // $\mathcal{F}$-NIZK asks for witness extraction

11: Parse $(eid, \sigma_\pi) \leftarrow \pi$
12: **assert** $\mathcal{G}$-Att$^{\mathrm{obsprog}} :: \mathcal{P}.\textsc{Verify}_{sid}(\sigma, (sid, eid, \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}, x)) = 1$
13: **if** $\exists w : (x, w) \in \Pi$ **then** // Adversary queried to prove x
14: $\quad$ **return** w
15: $\mathsf{Obs} \leftarrow \mathcal{G}$-Att$^{\mathrm{obsprog}} :: \mathcal{P}.\textsc{Observe}_{sid}()$
16: **if** $\exists w : (\textsc{resume}, (x, w), \cdot) \in \mathsf{Obs} \wedge (x, w) \in \mathcal{R}$ **then** // Environment queried to prove x
17: $\quad$ **return** w

[**Jan:** This figure is incomplete, in the sense that maybe other (trivial?) real-world adversary behavior isn't modeled.]

[**Jan:** This is not a reduction. Everything is information-theoretical. We're just arguing that both worlds produce the same view for the environment. I would disregard the below. We don't need game hops. Instead, I suggest we follow this structure:

(1) Argue that when env calls PROGRAM, ISPROGRAMMED, RESUME (via dummy adv or simulator), those behave the same in both worlds.
(2) Argue that NIZK::PROVE outputs a proof that looks exactly the same in both worlds.
(3) Argue that NIZK::VERIFY behaves the same in both worlds (i.e. that `Adv` :: EXTRACT never fails to output a valid witness)] [**Lorenzo:** fair] [**Lorenzo:** argue that (1) works because simulator replaces IsProgrammed in the ideal world with a version that omits simulated programming.] [**Lorenzo:** (2) holds because distribution of enclave ids is the same, and anon attestation if we use proxy] [**Lorenzo:** 3 holds because either the adversary proved through Gatt in this session (in which we case we store $(x, w) \in \Pi$), or because we can observe programming / resumptions in the wrong session]

■

6 Outsourcing in an observable world

We provide a protocol to implement NIZKs[R] using the passgatt functionality. The protocol UC emulates the weak NIZK functionality of [BFKT24] (functionality 3).

The protocol proceeds between three parties: Prover, Verifier, and Server.

On input Setup, Server installs program $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$ on modgatt and resumes $pk \leftarrow \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}(Init)$ which is sent to prover and verifier along with the attestation signature. Pr and Ver verify the attestation

On input Prove,x,w from Z, Prover sends Enc(pk,(x,w,crs)) to server. Server resumes $x, bit \leftarrow \mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}(Prove, ct)$ and sends attestation signature to Prover

Verifier just checks attestation

rescue protocol 1 but with right functionality

Protocol 5: Π^{V}-NIZK[$\mathcal{S}$] with server

$\mathcal{S}.\text{INIT}_{sid}()$ [**Jan:** Run before anything else?! Party-specific. Ordering of all inits? Not great.]

10: $eid \leftarrow \mathcal{G}\text{-Att}^{\text{mod}} :: \mathcal{S}.\text{INSTALL}_{sid}(sid, \mathsf{prog}_{\text{ZK}[\mathcal{R}]})$ [**Jan:** Longer *sid*? @MK?]
11: $(pk, \sigma) \leftarrow \mathcal{G}\text{-Att}^{\text{mod}} :: \mathcal{S}.\text{RESUME}(eid, Init)$
12: $\mathcal{F}\text{-Store} :: \mathcal{S}.\text{STORE}(pk, \sigma)$
13: **while** true **do**
14: $\quad ct \leftarrow \mathcal{F}\text{-Channel}[\mathcal{P}, \mathcal{S}] :: \mathcal{S}.\text{RECEIVE}()$
15: $\quad ((x, crs), \sigma) \leftarrow \mathcal{G}\text{-Att}^{\text{mod}} :: \mathcal{S}.\text{RESUME}(eid, \text{PROVE}, ct)$
16: $\quad \mathcal{F}\text{-Channel}[\mathcal{S}, \mathcal{P}] :: \mathcal{S}.\text{SEND}(\sigma)$

[**Jan:** General note/concern:
We imagine that Adv blocks Prove requests until the server is activated and processes the request. How does Adv notice when the server gets the activation token in the ideal world? Maybe we should amend $\mathcal{F}$-NIZK to include some Server::Process() explicitly. We may be stressing the boundaries of original $\mathcal{F}$-NIZK too much.]
[**Jan:** One potential (Canetti, not SSP-UC) solution: Have F-Channel in real world ask the adversary which message to deliver next, then have F-Channel pass activation to the recipient. That way, simulator can notice the activation in ideal world by simulating that interaction between Adv and F-Channel.]

$\mathcal{P}.\text{PROVE}(x, w)$
17: $crs \leftarrow \mathcal{F}\text{-CRS} :: \mathcal{P}.\text{CRS}()$
18: **repeat**
19: $\quad (pk, \sigma) \leftarrow \mathcal{F}\text{-Store} :: \mathcal{P}.\text{RETRIEVE}()$
20: $\quad$ [**Jan:** idk ... feels weird.]
21: **until** $\mathcal{G}\text{-Att}^{\text{mod}} :: \mathcal{V}.\text{VERIFY}(\sigma, crs) = 1$ [**Jan:** σ is on more than just *crs*]
22: $ct \leftarrow pke.enc(pk, (x, w, crs))$
23: **repeat**
24: $\quad \mathcal{F}\text{-Channel}[\mathcal{P}, \mathcal{S}] :: \mathcal{P}.\text{SEND}(ct)$
25: $\quad \sigma \leftarrow \mathcal{F}\text{-Channel}[\mathcal{S}, \mathcal{P}] :: \mathcal{P}.\text{RECEIVE}()$
26: **until** $\mathcal{G}\text{-Att}^{\text{mod}} :: \mathcal{V}.\text{VERIFY}(\sigma, (ct, x, crs)) = 1$
27: **return** $\pi = (ct, \sigma)$

$\mathcal{V}.\text{VERIFY}(x, \pi = (ct, \sigma))$
28: $b \leftarrow \mathcal{G}\text{-Att}^{\text{mod}} :: \mathcal{V}.\text{VERIFY}(\sigma, (ct, x, crs))$
29: **return** b

Program 6: $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$

INIT
5: $(pk, sk) \leftarrow PKE.KGen(1^\lambda)$
6: **return** pk

PROVE(ct)
7: $(x, w, crs) \leftarrow PKE.Dec(sk, ct)$
8: **assert** $(x, w) \in \mathcal{R}$[**Lorenzo:** gatt semantics: if assert failed, return bot with or without attestation?]
9: **return** (ct, x, crs)

SIMULATE(x, crs, τ)
10: **assert** $crs = \mathsf{CRS}(\tau)$
11: make up ct [**Jan:** todo]
12: **return** (ct, x, crs)

EXTRACT(ct, τ) [**Jan:** Someone (server?) is always corrupted (statically at the start), simulator installs TEE on that party.]
13: $(x, w, crs) \leftarrow PKE.Dec(sk, ct)$
14: **assert** $crs = \mathsf{CRS}(\tau)$
15: **return** (x, w)

Program $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$ is defined as follows: on input Init generate PKE parameters and return pk. On input prove,ct; decrypt ct into x,w,crs; return $x, \perp$ if $(x, w) \notin R$, otherwise return $x, \top, crs$. On input Simulate,x,tau return x,top,crs. On input Extract(ct,tau), decrypt ct to (x,w,crs). If crs = tau return x,w

//Variant On input $Verify, x, \pi$, verifies that π is a valid attestation output from this same enclave with return value x,top and returns whether that is true or not. add table check? Alt2: prove operation outputs ct for witness which is included as part of π, verification asserts witness is valid)

non-provable version is just as good?

6.1 Simulator

S initialises modgatt with some signature scheme (we could start with pass gatt signature scheme) for which it has the secret key. On receiving a simulation request from fwnizk, it uses the signature to produce an attesation over $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$ issuing x,top (problem: if x,w not in R, ideal world does not trigger simulator; if prover is honest, we will not activate, but real world server should obtain an attestation over x,bot. Fix: leaky channel tells you that the prover fails, and I can return bot attestation) and store x.

When receiving an extract message including statement and proof, if x was stored send maul signature? what is the role of the extracted witness?)

If sigma is weaker, it is possible to provide signatures that attest to some F(meas) ->? mauling?

7 Transparent enclaves

[**Markulf:** Expand to other attacks and add combiners] [TZL+16] define a notion of Transparent enclaves. Describe. Definitional Issues: returns randomness to environment. Sealed glass-proof.

Semi-transparent enclaves [DMM23]

[**Markulf:** Outsourced NIZK computation in TEE is the combiner IOG is interested in.]

7.1 Revising transparency definition

Two versions of Transparency depending on where the leakage goes to. Use direct leak interface to simulator

Functionality 6: $\mathcal{G}\text{-}\mathtt{Att}^{\text{Transp}}[\lambda, \mathsf{reg}, \Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})]$ [PST17]

$\underline{\text{INIT}_{sid}}$
1: $(pk_{\text{att}}, sk_{\text{att}}) \leftarrow \mathsf{Gen}(1^\lambda)$
2: $\mathsf{Sig}[\cdot] \leftarrow \bot$
3: $\mathsf{Encl}[\cdot,\cdot] \leftarrow \bot$

$\underline{\mathcal{P}.\text{GETPK}_{sid}()}$ for all $\mathcal{P}, sid$
4: **return** pk_{att}

$\underline{\mathcal{P}.\text{VERIFY}_{sid}(\sigma, meas)}$ for all $\mathcal{P}, sid$
5: **return** $meas \overset{?}{\in} \mathsf{Sig}[\sigma]$

$\underline{\mathcal{P}.\text{LEAK}_{sid}(eid)}$ for corrupted $\mathcal{P} \in \mathsf{reg}$
6: // Need to enforce that the interface is adversarial
7: fetch $(\cdot,\cdot,r) \leftarrow \mathsf{Encl}[eid, \mathcal{P}]$ [**Jan:** third component is *st* right now, not random string. Intended?]
8: **return** r

[**Jan:** Question: when I corrupt a party, do I learn its TEE's previous randomness? Or just future randomness? (i.e. do TEEs have secure erasure of randomness?)]

$\underline{\mathcal{P}.\text{INSTALL}_{sid}(idx, \mathsf{prog})}$ for $\mathcal{P} \in \mathsf{reg}$ and all sid
9: $eid \xleftarrow{\$} \{0,1\}^\lambda$
10: **if** $\mathcal{P}$ is not corrupted **then**
11: $\quad idx \leftarrow sid$
12: $\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st = ())$
13: **return** eid

$\underline{\mathcal{P}.\text{RESUME}_{sid}(eid, input)}$ for $\mathcal{P} \in \mathsf{reg}$ and all sid
14: **assert** $\mathsf{Encl}[eid, \mathcal{P}] \neq \bot$
15: $(\mathsf{prog}, idx, st, r) \leftarrow \mathsf{Encl}[eid, \mathcal{P}]$
16: $(output, st', r') \leftarrow \mathsf{prog}(input, st, r)$
17: $\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st', r')$
18: $meas \leftarrow (idx, eid, \mathsf{prog}, output)$
19: $\sigma \leftarrow \mathsf{Sign}_{sk_{\text{att}}}(meas)$
20: $\mathsf{Sig}[\sigma] \leftarrow \mathsf{Sig}[\sigma] \cup \{meas\}$
21: **return** $(output, \sigma)$

$\mathcal{G}\text{-}\mathtt{Att}^{\text{FullTransp}}$ is defined as above except that LEAK does not require $\mathcal{P}$ to be corrupted, and additionally returns inputs (which need to be stored on the table - do we need to store all inputs as in FullyTransparent gatt)?

7.2 Revising Sealed-Glass Proofs with programmability

7.3 Advanced ZKP for Full transparency

Acknowledgment

References

BFKT24. J. Bobolz, P. Farshim, M. Kohlweiss, and A. Takahashi. The brave new world of global generic groups and UC-secure zero-overhead SNARKs. Cryptology ePrint Archive, Report 2024/818, 2024.

Can20. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Paper 2000/067, 2020. https://eprint.iacr.org/2000/067.

CDG+18. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In *EUROCRYPT 2018, Part I*, vol. 10820 of *LNCS*, pp. 280–312. Springer, Cham, 2018.

DMM23. F. Dörre, J. Mechler, and J. Müller-Quade. Practically efficient private set intersection from trusted hardware with side-channels. In *ASIACRYPT 2023, Part IV*, vol. 14441 of *LNCS*, pp. 268–301. Springer, Singapore, 2023.

PST17. R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. In *EUROCRYPT 2017, Part I*, vol. 10210 of *LNCS*, pp. 260–289. Springer, Cham, 2017.

TZL+16. F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016.

A TEE Functionalities

[**Jan:** These are unfinished (semi-discarded?) sketches.]

Functionality 7: $\mathcal{G}\text{-}\mathtt{Att}^{\text{mod}}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}]$

INITIALISE

$(pk_{\text{att}}, \mathsf{Sign}) \leftarrow \mathtt{Adv} :: \textsc{Initialize}()$
$\mathsf{Sig}[\cdot] \leftarrow \bot$
$\mathsf{Encl}[\cdot,\cdot] \leftarrow \bot$

$\mathcal{P}.\textsc{getpk}_{sid}()$ for all $\mathcal{P}, sid$

return pk_{att}

$\mathcal{P}.\textsc{install}_{sid}(idx, \mathsf{prog})$ for $\mathcal{P} \in \mathsf{reg}$ and all sid

if $\mathcal{P}$ is not corrupted **then**
  assert $idx = sid$
for instruction $i \in \mathsf{prog}$ **do**
  if $i \notin \mathbb{O}$ **then**
    return MissingInstructionError
$eid \xleftarrow{\$} \{0,1\}^\lambda$
$\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (idx, \mathsf{prog})$
Send install to $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (eid||\mathcal{P}, \text{“att”}||idx))$ [**Jan:** How to formalize in this paper?]
return eid

$\mathcal{P}.\textsc{resume}_{sid}(eid, input, attack)$ for $\mathcal{P} \in \mathsf{reg}$ and all sid

assert $\mathsf{Encl}[eid, \mathcal{P}] \neq \bot$
$(idx, \mathsf{prog}) \leftarrow \mathsf{Encl}[eid, \mathcal{P}]$
if $attack = \epsilon$ or $\mathcal{P}$ is not corrupted **then**
  Send $input$ to $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (eid||\mathcal{P}, \text{“att”}||idx))$ and receive $output$
else
  assert $attack \in \mathbb{A}$
  Send $attack, input$ to $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (eid||\mathcal{P}, \text{“att”}||idx))$ and receive $output, aux$
  if $aux \neq \epsilon$ **then**
    $\mathtt{Adv} :: \mathcal{P}.\textsc{Resume}_{sid}(attack, aux)$
$meas \leftarrow \mathbb{S}(\text{configuration of } \text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}])$
$\sigma \leftarrow \mathsf{Sign}(meas)$
$\mathsf{Sig}[\sigma] \leftarrow \mathsf{Sig}[\sigma] \cup \{meas\}$
return $(output, \sigma)$

$\mathcal{P}.\textsc{verify}(\sigma, meas)$ for all $\mathcal{P}, sid$

return $meas \stackrel{?}{\in} \mathsf{Sig}[\sigma]$

Functionality 8: $\mathcal{G}\text{-}\mathtt{Att}^{\text{mod}}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}]$ de-Turing-machined (suggestion)

INITIALISE

$(pk_{\text{att}}, \mathsf{Sign}) \leftarrow \mathtt{Adv} :: \textsc{Initialize}()$
$\mathsf{Sig}[\cdot] \leftarrow \bot$
$\mathsf{Encl}[\cdot,\cdot] \leftarrow \bot$

$\mathcal{P}.\textsc{getpk}_{sid}()$ for all $\mathcal{P}, sid$

return k

$\mathcal{P}.\textsc{install}_{sid}(idx, \mathsf{prog})$ for $\mathcal{P} \in \mathsf{reg}$ and all sid

assert $\mathsf{prog} \in \mathbb{P}(\mathbb{O})$
$eid \xleftarrow{\$} \{0,1\}^\lambda$
if $\mathcal{P}$ is not corrupted **then**
  $idx \leftarrow sid$
$\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st_0 = ())$
return eid

$\mathcal{P}.\textsc{verify}(\sigma, meas)$ for all $\mathcal{P}, sid$

return $meas \stackrel{?}{\in} \mathsf{Sig}[\sigma]$

$\mathcal{P}.\textsc{resume}_{sid}(eid, input, attack)$ for $\mathcal{P} \in \mathsf{reg}$ and all sid

assert $\mathsf{Encl}[eid, \mathcal{P}] \neq \bot$
$(\mathsf{prog}, idx, st_0, \ldots, st_n) \leftarrow \mathsf{Encl}[eid, \mathcal{P}]$
if $\mathcal{P}$ is corrupted $\wedge attack \in \mathbb{A}$ **then**
  $(st_{n+1}, output, meas, aux) \leftarrow attack^{\mathbb{O}}(\mathcal{P}, eid, idx, \mathsf{prog}, (st_0, \ldots, st_n), input)$
else
  $(st_{n+1}, output) \leftarrow \mathsf{prog}^{\mathbb{O}}(st_n, input)$
  $meas \leftarrow \mathbb{S}(\mathcal{P}, eid, idx, \mathsf{prog}, (st_0, \ldots, st_{n+1}), output)$
  $aux \leftarrow \epsilon$
$\mathsf{Encl}[eid, \mathcal{P}] \leftarrow (\mathsf{prog}, idx, st_0, \ldots, st_n, st_{n+1})$
if $aux \neq \epsilon$ **then** [**Jan:** maybe remove]
  $\mathtt{Adv} :: \mathcal{P}.\textsc{resume}_{sid}(attack, aux)$ // Delay
// Attest to measurement $meas$
$\sigma \leftarrow \mathsf{Sign}(meas)$
$\mathsf{Sig}[\sigma] \leftarrow \mathsf{Sig}[\sigma] \cup \{meas\}$
return $(output, meas, \sigma)$

[**Jan:** $\mathbb{O}$ contains stuff like: Encrypt/Decrypt (for transparent), mUC F-NIZK calls for ZK relation $\mathcal{R}$]

B Programmability in AGATE

We augment the modgatt functionality with a new argument f that a malicious party can provide during installation. f is a function that modifies the attestation measurement to be signed as part of a resume operation i.e. after a resume, the modgatt functionality returns $output, \mathbb{S}(f(meas))$ instead of $output, \mathbb{S}(meas)$. In our case, f is a function that replaces the code of some program $\bar{ZK}[R]$ attested in $meas$ value with program $\hat{ZK}[R]$. When a party calls verify on some attestation σ for message m, Verify will also return the value of f if it has been set for the enclave that produced σ, and the verification caller is in the current session

A real world (gatt) verifier will thus always learn if a malicious party has installed a different program than what they claim their attestation to be for. In the ideal world, the simulator is able to trick malicious verifiers by omitting to pass on f whenever the gatt verify subroutine is called. In both worlds, protocols external to the sessions learn nothing about the programming

$\hat{ZK}[R]$ is the same as $\mathsf{prog}_{\mathrm{ZK}[\mathcal{R}]}$ without the CRS (define). $\bar{ZK}[R]$ is defined as the program $\hat{ZK}[R]$ without the extract and simulate subroutines

This version of the programmable gatt is more general then adding a Program/IsProgrammed interface as it lets us define adaptive programming based on the internal values of the enclave and does not require the simulator to run the enclave code in its head.