Financially Backed Covert Security

Sebastian Faust[1], Carmit Hazay[2], David Kretzler[1], and Benjamin Schlosser[1]

[1] Technical University of Darmstadt, Germany
{first.last}@tu-darmstadt.de
[2] Bar-Ilan University, Israel
carmit.hazay@biu.ac.il

Abstract. The security notion of *covert security* introduced by Aumann and Lindell (TCC'07) allows the adversary to successfully cheat and break security with a fixed probability $1 - \epsilon$, while with probability ϵ, honest parties detect the cheating attempt. Asharov and Orlandi (ASIACRYPT'12) extend covert security to enable parties to create publicly verifiable evidence about misbehavior that can be transferred to any third party. This notion is called *publicly verifiable covert security* (PVC) and has been investigated by multiple works. While these two notions work well in settings with known identities in which parties care about their reputation, they fall short in Internet-like settings where there are only digital identities that can provide some form of anonymity.
In this work, we propose the notion of *financially backed covert security* (FBC), which ensures that the adversary is financially punished if cheating is detected. Next, we present three transformations that turn PVC protocols into FBC protocols. Our protocols provide highly efficient judging, thereby enabling practical judge implementations via smart contracts deployed on a blockchain. In particular, the judge only needs to non-interactively validate a single protocol message while previous PVC protocols required the judge to emulate the whole protocol. Furthermore, by allowing an interactive punishment procedure, we can reduce the amount of validation to a single program instruction, e.g., a gate in a circuit. An interactive punishment, additionally, enables us to create financially backed covert secure protocols without any form of common public transcript, a property that has not been achieved by prior PVC protocols.

Keywords: Covert Security · Multi-Party Computation (MPC) · Public Verifiability · Financial Punishment

1 Introduction

Secure multi-party computation (MPC) protocols allow a set of parties to jointly compute an arbitrary function f on private inputs. These protocols guarantee privacy of inputs and correctness of outputs even if some of the parties are corrupted by an adversary. The two standard adversarial models of MPC are *semi-honest* and *malicious* security. While semi-honest adversaries follow the protocol description but try to derive information beyond the output from the

interaction, malicious adversaries can behave in an arbitrary way. MPC protocols in the malicious adversary model provide stronger security guarantees at the cost of significantly less efficiency. As a middle ground between good efficiency and high security Aumann and Lindell introduced the notion of *security against covert adversaries* [AL07]. As in the malicious adversary model, corrupted parties may deviate arbitrarily from the protocol specification but the protocol ensures that cheating is detected with a fixed probability, called *deterrence factor* ϵ. The idea of covert security is that adversaries fear to be detected, e.g., due to reputation issues, and thus refrain from cheating.

Although cheating can be detected in covert security, a party of the protocol cannot transfer the knowledge about malicious behavior to other (external) parties. This shortcoming was addressed by Asharov and Orlandi [AO12] with the notion of *covert security with public verifiability* (PVC). Informally, PVC enables honest parties to create a publicly verifiable certificate about the detected malicious behavior. This certificate can subsequently be checked by any other party (often called *judge*), even if this party did not contribute to the protocol execution. The idea behind this notion is to increase the deterrent effect by damaging the reputation of corrupted parties publicly. PVC secure protocols for the two-party case were presented by [AO12, KM15, ZDH19, HKK$^+$19]. Recently, Damgård et al. [DOS20] showed a generic compiler from semi-honest to publicly verifiable covert security for the two-party setting and gave an intuition on how to extend their compiler to the multi-party case. Full specifications of generic compilers from semi-honest to publicly verifiable covert security for multi-party protocols were presented by Faust et al. [FHKS21] and Scholl et al. [SSS21].

Although PVC seems to solve the shortcoming of covert security at first glance, in many settings PVC is not sufficient; especially, if only a digital identity of the parties is known, e.g., in the Internet. In such a setting, a real party can create a new identity without suffering from a damaged reputation in the sequel. Hence, malicious behavior needs to be punished in a different way. A promising approach is to use existing cryptocurrencies to directly link cheating detection to financial punishment without involving trusted third parties; in particular, cryptocurrencies that support so-called *smart contracts*, i.e., programs that enable the transfer of assets based on predefined rules. Similar to PVC, where an external judge verifies cheating by checking a certificate of misbehavior, we envision a smart contract that decides whether a party behaved maliciously or not. In this setting, the task of judging is executed over a distributed blockchain network keeping it incorruptible and verifiable at the same time. Since every instruction executed by a smart contract costs fees, it is highly important to keep the amount of computation performed by a contract small. This aspect is not solely important for execution of smart contracts but in all settings where an external judge charges by the size of the task it gets. Due to this constraint, we cannot straightforward adapt PVC protocols to work in this setting, since detection of malicious behavior in existing PVC protocols is performed in a naive way that requires the judge to recompute a whole protocol execution.

Related work. While combining MPC with blockchain technologies is an active research area (e.g., [KB14, BK14, ADMM14]) none of these works deal with realizing the judging process of PVC protocols over a blockchain. The only work connecting covert security with financial punishment thus far is by Zhu et al. [ZDH19], which we describe in a bit more detail below. They combine a two-party garbling protocol with an efficient judge that can be realized via a smart contract. Their construction leverages strong security primitives, like a malicious secure oblivious transfer for the transmission of input wires, to ensure that cheating can only occur during the transmission of the garbled circuit and not in any other part of the two-party protocol. By using a binary search over the transmitted circuit, the parties narrow down the computation step under dispute to a single circuit gate. This process requires $O(\log(|C|))$ interactions, where $|C|$ denotes the circuit size, and enables the judge to resolve the dispute by recomputing only a single circuit gate.

While the approach of Zhu et al. [ZDH19] provides an elegant way to reduce the computational complexity of the judge in case cheating is restricted to a single message, it falls short if multiple messages or even a whole protocol execution is under dispute. As a consequence, their construction is limited in scalability and generality, since it is only applicable to two-party garbling protocols, i.e., neither other semi-honest two-party protocols nor more parties are supported.

Generalizing the ideas of [ZDH19] to work for other protocol types and the multi-party case requires us to address several challenges. First, in [ZDH19] the transmitted garbled circuit under dispute is the result of the completely non-interactive garbling process. In contrast, many semi-honest MPC protocols (e.g., [GMW87, BMR90]) consist of several rounds of interactions that need to be all considered during the verification. Interactivity poses the challenge that multiple messages may be under dispute and the computation of messages performed by parties may depend on data received in previous rounds. Hence, verifications of messages need to consider local computations and internal states of the parties that depend on all previous communication rounds. This task is far more complex than verifying a single public message. Second, supporting more than two parties poses the challenge of resolving a dispute about a protocol execution during which parties might not know the messages sent between a subset of other parties. Third, the transmitted garbled circuit in [ZDH19] is independent of the parties private inputs. Considering protocols where parties provide secret inputs or messages that depend on these inputs, requires a privacy-preserving verification mechanism to protect parties' sensitive data.

1.1 Contribution

Our first contribution is to introduce a new security notion called *financially backed covert security* (FBC). This notion combines a covertly secure protocol with a mechanism to financially punish a corrupted party if cheating was detected. We formalize financial security by adding two properties to covert security, i.e., *financial accountability* and *financial defamation freeness*. Our notion is similar to the one of PVC; in fact, PVC adds reputational punishment

to covert security via *accountability* and *defamation freeness*. In order to lift these properties to the financial context, FBC requires deposits from all parties and allows for an interactive judge. We present two security games to formalize our introduced properties. While the properties are close to accountability and defamation freeness of PVC, our work for the first time explicitly presents formal security games for these security properties, thereby enabling us to rigorously reason about financial properties in PVC protocols. We briefly compare our new notion to the security definition of Zhu et al. [ZDH19], which is called *financially secure computation*. Zhu et al. follow the approach of simulation-based security by presenting an ideal functionality for two parties that extends the ideal functionality of covert security. In contrast, we present a game-based security definition that is not restricted to the two-party case. While simulation-based definitions have the advantage of providing security under composition, proving a protocol secure under their notion requires to create a full simulation proof which is an expensive task. Instead, our game-based notion allows to re-use simulation proofs of all existing covert and PVC protocols, including future constructions, and to focus on proving financial accountability and financial defamation freeness in a standalone way.

We present transformations from different classes of PVC protocols to FBC protocols. While we could base our transformations on covert protocols, FBC protocols require a property called *prevention of detection dependent abort*, which is not always guaranteed by a covert protocol. The property ensures that a corrupted party cannot abort after learning that her cheating will be detected without leaving publicly verifiable evidence. PVC protocols always satisfy prevention of detection dependent abort. So, by basing our transformation on PVC protocols, we inherit this property.

While the mechanism utilized by [ZDH19] to validate misbehavior is highly efficient, it has only been used for non-interactive algorithms so far, i.e, to validate correctness of the garbling process. We face the challenge of extending this mechanism over an interactive protocol execution while still allowing for efficient dispute resolution such that the judge can be realized via a smart contract. In order to tackle these challenges, we present a novel technique that enables efficient validation of arbitrary complex and interactive protocols given the randomness and inputs of all parties. What's more, we can allow for private inputs if a public transcript of all protocol messages is available. We utilize only standard cryptographic primitives, in particular, commitments and signatures.

We differentiate existing PVC protocols according to whether the parties provide private inputs or not. The former protocols are called *input-dependent* and the latter ones *input-independent*. Input-independent protocols are typically used to generate correlated randomness. Further, all existing PVC protocols incorporate some form of common public transcript. Input-dependent protocols require a common public transcript of messages. In contrast, for input-independent protocols, it is enough to agree on the hashes of all sent messages. While it is not clear, if it is possible to construct PVC protocols without any form of public transcript, we construct FBC protocols providing this property. We achieve this

by exploiting the interactivity of the judge, which is non-interactive in PVC. Based on the above observations, we define the following three classes of FBC protocols, for which we present transformations from PVC protocols.

Class 1: The first class contains *input-independent* protocols during which parties learn hashes of all protocol messages such that they agree on a common *transcript of message hashes*.

Class 2: The second class contains *input-dependent* protocols with a public *transcript of messages*. In contrast to class 1, parties may provide secret inputs and share a common view on all messages instead of a common view on hashes only.

Class 3: The third class contains input-independent protocols where parties do not learn any information about messages exchanged between a subset of other parties (cf. class 1). As there are no PVC protocol fitting into this class, we first convert PVC protocols matching the requirements of class 1 into protocols without public transcripts and second leverage an interactive punishment procedure to transform the resulting protocols into FBC protocols without public transcripts. Our FBC protocols benefit from this property since parties have to send all messages only to the receiver and not to all other parties. This effectively reduces the concrete communication complexity by a factor depending on the number of parties. In the optimistic case, if there is no cheating, we get this benefit without any overhead in the round complexity.

For each of our constructions, we provide a formal specification and a rigorous security analysis; the ones of the second class can be found in the supplementary materials F. This is in contrast to the work of [ZDH19] which lacks a formal security analysis for financially secure computation. We stress that all existing PVC multi-party protocols can be categorized into class 1 and 2. Additionally, by combining any of the transformations from [DOS20, FHKS21, SSS21], which compile semi-honest protocols into PVC protocols, our constructions can be used to transform these protocol into FBC protocols.

The resulting FBC protocols for class 1 and 2 allow parties to non-interactively send evidence about malicious behavior to the judge. As the judge entity in these two classes is non-interactive, techniques from our transformations are of independent interest to make PVC protocols more efficient. Since, in contrast to class 1 and 2, there is no public transcript present in protocols of class 3, we design an interactive process involving the judge entity to generate evidence about malicious behavior. For all protocols, once the evidence is interactively or non-interactively created, the judge can efficiently resolve the dispute by recomputing only a single protocol message regardless of the overall computation size. We can further reduce the amount of validation to a single program instruction, e.g., a gate in a circuit, by prepending an interactive search procedure. This extension is presented in the supplementary materials G.

Finally, we provide a smart contract implementation of the judging party in Ethereum and evaluate its gas costs (cf. Section 8). The evaluation shows the

practicability, e.g., in the three party setting, with optimistic execution costs of 533 k gas. Moreover, we show that the dispute resolution of our solution is highly scalable in regard to the number of parties, the number of protocol rounds and the protocol complexity.

1.2 Technical Overview

In this section, we outline the main techniques used in our work and present the high-level ideas incorporated into our constructions. We start with on overview of the new notion of *financially backed covert security*. Then, we present a first attempt of a construction over a blockchain and outline the major challenges. Next, we describe the main techniques used in our constructions for PVC protocols of classes 1 and 2 and finally elaborate on the bisection procedure required for the more challenging class 3.

Financially backed covert security. We recall that, a publicly verifiable covertly secure (PVC) protocol $(\pi_{\mathsf{cov}}, \mathsf{Blame}, \mathsf{Judge})$ consists of a covertly secure protocol π_{cov}, a blaming algorithm Blame and a judging algorithm Judge. The blaming algorithm produces a certificate cert in case cheating was detected and the judging algorithm, upon receiving a valid certificate, outputs the identity of the corrupted party. The algorithm Judge of a PVC protocol is explicitly defined as non-interactive. Therefore, cert can be transferred at any point in time to any third party that executes Judge and can be convinced about malicious behavior if the algorithm outputs the identity of a corrupted party.

In contrast to PVC, *financially backed covert security* (FBC) works in a model where parties own assets which can be transferred to other parties. This is modelled via a ledger entity $\mathcal{L}$. Moreover, the model contains a trusted judging party $\mathcal{J}$ which receives deposits before the start of the protocol and adjudicates in case of detected cheating. We emphasize that the entity $\mathcal{J}$, which is a single trusted entity interacting with all parties, is not the same as the algorithm Judge of a PVC protocol, which can be executed non-interactively by any party. An FBC protocol $(\pi'_{\mathsf{cov}}, \mathsf{Blame}', \mathsf{Punish})$ consists of a covertly secure protocol π'_{cov}, a blaming algorithm Blame' and an interactive punishment protocol Punish. Similar to PVC, the blaming algorithm Blame' produces a certificate cert' that is used as an input to the interactive punishment protocol. Punish is executed between the parties and the judge $\mathcal{J}$. If all parties behave honestly during the execution of π'_{cov}, $\mathcal{J}$ sends the deposited coins back to all parties after the execution of Punish. In case cheating is detected during π'_{cov}, the judge $\mathcal{J}$ burns the coins of the cheating party.

First attempt of an instantiation over a blockchain. Blockchain technologies provide a convenient way of handling monetary assets. In particular, in combination with the execution of smart contracts, e.g., offered by Ethereum [W+14], we envision to realize the judging party $\mathcal{J}$ as a smart contract. A first attempt of designing the punishment protocol is to implement $\mathcal{J}$ in a way, that the judge just gets the certificate generated by the PVC protocol's blame algorithm and

executes the PVC protocol's Judge-algorithm. However, the Judge-algorithm of all existing PVC protocols recomputes a whole protocol instance and compares the output with a common transcript on which all parties agree beforehand. As computation of a smart contract costs money in form of transaction fees, recomputing a whole protocol is prohibitively expensive. Therefore, instead of recomputing the whole protocol, we aim for a punishment protocol that facilitates a judging party $\mathcal{J}$ which needs to recompute just a single protocol step or even a single program instruction, e.g., a gate in a circuit. The resulting judge becomes efficient in a way that it can be practically realized via a smart contract.

FBC protocols with efficient judging from PVC protocols. In this work, we present three transformations from PVC protocols to FBC protocols. Our transformations start with PVC protocols providing different properties which we use to categorize these protocols into three classes. We model the protocol execution in a way such that every party's behavior is deterministically defined by her input, her randomness and incoming messages. More precisely, we define the initial state of a party as her input and some randomness and compute the next state according to the state of the previous round and the incoming messages of the current round. Our first two transformations build on PVC protocols where the parties share a public transcript of the exchanged messages resp. message hashes. Additionally, parties send signed commitments on their intermediate states to all parties. The opening procedure ensures that correctly created commitments can be opened – falsely created commitments open to an invalid state that is interpreted as an invalid message. By sending the internal state of some party P_m for a single round together with the messages received by P_m in the same round to the judging party, the latter can efficiently verify malicious behavior by recomputing just a single protocol step. The resulting punishment protocol is efficient and can be executed without contribution of the cheating party.

Interactive punishment protocol to support private transcripts. Our third transformation compiles input-independent PVC protocols with a public transcript into protocols where no public transcript is known to the parties. The lack of a public transcript makes the punishment protocol more complicated. Intuitively, since an honest party has no signed information about the message transcript, she cannot provide verifiable data about the incoming message used to calculate a protocol step. Therefore, we use the technique of an interactive bisection protocol which was first used in the context of verifiable computing by Canetti et al. [CRR11] and subsequently by many further constructions [KGC+18, TR19, ZDH19, EFS20]. While the bisection technique is very efficient to narrow down disagreement, it was only used for non-interactive algorithms so far. Hence, we extend this technique to support also interactive protocols. In particular, in our work, we use a bisection protocol to allow two parties to efficiently agree on a common message history. To this end, both parties, the accusing and the accused one, create a Merkle tree of their emulated message history up to the disputed message and submit the corresponding root. If they agree on the message history, the accusation can be validated by ref-

erence to this history. If they disagree, they perform a bisection search over the proposed history that determines the first message in the message history, they disagree on, while automatically ensuring that they agree on all previous messages. Hence, the judge can verify the message that the parties disagree on based on the previous messages they agree on. At the end of both interactions, the judge can efficiently resolve the dispute by recomputing just a single step.

2 Preliminaries

We start by introducing notation and cryptographic primitives used in our construction. Moreover, we provide the definition of covert security and publicly verifiable covert security in the supplementary materials A and B.

We denote the computational security parameter by κ. Let n be some integer, then $[n] = \{1, \ldots, n\}$. Let $i \in [n]$, then we use the notation $j \neq i$ for $j \in [n] \setminus \{i\}$. A function $\mathsf{negl}(n) : \mathbb{N} \to \mathbb{R}$ is *negligible* in n if for every positive integer c there exists an integer n_0 such that $\forall n > n_0$ it hols that $\mathsf{negl}(n) < \frac{1}{n^c}$. We use the notation $\mathsf{negl}(n)$ to denote a negligible function.

We define $\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa)$ to be the output of the execution of an n-party protocol π executed between parties $\{P_i\}_{i\in[n]}$ on input $\bar{x} = \{x_i\}_{i\in[n]}$ and security parameter κ, where $\mathcal{A}$ on auxiliary input z corrupts parties $\mathcal{I} \subset \{P_i\}_{i\in[n]}$. We further specify $\mathsf{OUTPUT}_j(\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa))$ to be the output of party P_j for $j \in [n]$.

Our protocol utilizes a signature scheme $(\mathsf{Generate}, \mathsf{Sign}, \mathsf{Verify})$ that is *existentially unforgeable under chosen-message attacks*. We assume that each party executes the $\mathsf{Generate}$-algorithm to obtain a key pair $(\mathsf{pk}, \mathsf{sk})$ before the protocol execution. Further, we assume that all public keys are published and known to all parties while the secret keys are kept private. To simplify the protocol description we denote signed messages with $\langle x \rangle_i$ instead of $(x, \sigma := \mathsf{Sign}_{\mathsf{sk}_i}(x))$. The verification is therefore written as $\mathsf{Verify}(\langle x \rangle_i)$ instead of $\mathsf{Verify}_{\mathsf{pk}_i}(x, \sigma)$. Further, we make use of a hash function $H(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa$ that is collision resistant.

We assume a synchronous communication model, where communication happens in rounds and all parties are aware of the current round. Messages that are sent in some round k arrive at the receiver in round $k+1$. Since we consider a rushing adversary, the adversary learns the messages sent by honest parties in round k in the same round and hence can adapt her own messages accordingly. We denote a message sent from party P_i to party P_j in round k of some protocol instance denoted with ℓ as $\mathsf{msg}_{(\ell,k)}^{(i,j)}$. The hash of this message is denoted with $\mathsf{hash}_{(\ell,k)}^{(i,j)} := H(\mathsf{msg}_{(\ell,k)}^{(i,j)})$.

A *Merkle tree* over an ordered set of elements $\{x_i\}_{i\in[N]}$ is a labeled binary hash tree, where the i-th leaf is labeled by x_i. We assume N to be an integer power of two. In case the number of elements is not a power of two, the set can be padded until N is a power of two. For construction of Merkle trees, we make use of the collision-resistant hash function $H(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa$.

Formally, we define a Merkle tree as a tuple of algorithms ($\mathsf{MTree}, \mathsf{MRoot}$, $\mathsf{MProof}, \mathsf{MVerify}$). Algorithm MTree takes as input a computational security parameter κ as well as a set of elements $\{x_i\}_{i\in[N]}$ and creates a Merkle tree mTree. To ease the notation, we will omit the security parameter and implicitly assume it to be provided. Algorithm MRoot takes as input a Merkle tree mTree and returns the root element root of tree mTree. Algorithm MProof takes as input a leaf x_j and Merkle tree mTree and creates a Merkle proof σ showing that x_j is the j-th leaf in mTree. Algorithm $\mathsf{MVerify}$ takes as input a proof σ, an index i, a root root and a leaf x^* and returns true iff x^* is the i-the leaf of a Merkle tree with root root.

A Merkle Tree satisfies the following two requirements. First, for each Merkle tree mTree created over an arbitrary set of elements $\{x_i\}_{i\in[N]}$, it holds that for each $j \in [N]$ $\mathsf{MVerify}(\mathsf{MProof}(x_j, \mathsf{mTree}), j, \mathsf{MRoot}(\mathsf{mTree}), x_j) = \mathsf{true}$. We call this property *correctness*. Second, for each Merkle tree mTree with root $\mathsf{root} := \mathsf{MRoot}(\mathsf{mTree})$ created over an arbitrary set of elements $\{x_i\}_{i\in[N]}$ with security parameter κ it holds that for each polynomial time algorithm adversary $\mathcal{A}$ outputting an index j^*, leaf $x^* \neq x_{j^*}$ and proof σ^* the probability that $\mathsf{MVerify}(\sigma^*, j^*, \mathsf{MRoot}(\mathsf{mTree}), x^*) = \mathsf{true}$ is $\mathsf{negl}(\kappa)$. We call this property *binding*.

3 Financially Backed Covert Security

In the following, we specify the new notion of *financially backed covert security*. This notion extends covert security by a mechanism of financial punishment. More precisely, once an honest party detects cheating of the adversary during the execution of the covertly secure protocol, there is some corrupted party that is financial punished afterwards. The financial punishment is realized by an interactive protocol Punish that is executed directly after the covertly secure protocol. In order to deal with monetary assets, financially backed covertly secure protocols depend on a public ledger $\mathcal{L}$ and a trusted judge $\mathcal{J}$. The former can be realized by distributed ledger technologies, such as blockchains, and the latter by a smart contract executed on the said ledger. In the following, we describe the role of the ledger and the judging party, formally define financially backed covert security and outline techniques to prove financially backed covert security.

3.1 The Ledger and Judge

An inherent property of our model is the handling of assets and asset transfers based on predefined conditions. Nowadays, distributed ledger technologies like blockchains provide convenient means to realize this functionality. We model the handling of assets resp. coins via a ledger entity denoted by $\mathcal{L}$. The entity stores a balance of coins for each party and transfers coins between parties upon request. More precisely, $\mathcal{L}$ stores a balance $b_i^{(t)}$ for each party P_i at time t. For the security definition presented in Section 3.2, we are in particular interested in the balances before the execution of the protocol π, i.e., $b_i^{(\mathsf{pre})}$, and after the

execution of the protocol Punish, i.e., $b_i^{(\mathsf{post})}$. The balances are public such that every party can query the amount of coins for any party at the current time. In order to send coins to another party, a party interacts with $\mathcal{L}$ to trigger the transfer.

While we consider the ledger as a pure storage of balances, we realize the conditional transfer of coins based on some predefined rules specified by the protocol Punish via a judge $\mathcal{J}$. In particular, $\mathcal{J}$ constitutes a trusted third party that interacts with the parties of the covertly secure protocol. More precisely, we require that each party sends some fixed amount of coins as deposit to $\mathcal{J}$ before the covertly secure protocol starts. During the covertly secure protocol execution, the judge keeps the deposited coins but does not need to be part of any interaction. After the execution of the covertly secure protocol, the judge plays an important role in the punishment protocol Punish. In case any party detects cheating during the execution of the covertly secure protocol, $\mathcal{J}$ acts as an adjudicator. If there is verifiable evidence about malicious behavior of some party, the judge financially punishes the corrupted party by withholding her deposit. Eventually, $\mathcal{J}$ will reimburse all parties with their deposits except those parties that have been proven to be malicious. The rules according to which parties are considered malicious and hence according to which the coins are reimbursed or withhold need to be specified by the protocol Punish.

Finally, we emphasize that both entities the ledger $\mathcal{L}$ and the judge $\mathcal{J}$ are considered trusted. This means, the correct functionality of these entities cannot be distorted by the adversary.

3.2 Formal Definition

We work in a model in which a ledger $\mathcal{L}$ and a judge $\mathcal{J}$ as explained above exist. Let π' be an n-party protocol that is covertly secure with deterrence factor ϵ. Let the number of corrupted parties that is tolerated by π' be $m < n$ and the set of corrupted parties be denoted by $\mathcal{I}$. We define π as an extension of π', in which all involved parties transfer a fixed amount of coins, d, to $\mathcal{J}$ before executing π'. Additionally, after the execution of π', all parties execute algorithm Blame which on input the view of the honest party outputs a certificate and broadcasts the generated certificate – still as part of π. The certificate is used for both proving malicious behavior, if detected, and defending against being accused for malicious behavior.

After the execution of π, all parties participate in the protocol Punish. In case honest parties detected misbehavior, they prove said misbehavior to $\mathcal{J}$ such that $\mathcal{J}$ can punish the malicious party. In case a malicious party blames an honest one, the honest parties participate to prove their correct behavior. Either way, even if there is no blame at all, all honest parties wait to receive their deposits back, which are reimbursed by $\mathcal{J}$ at the end of the punishment protocol Punish.

Definition 1 (Financially backed covert security). *We call a triple* $(\pi, \mathsf{Blame}, \mathsf{Punish})$ *an n-party* financially backed covertly secure protocol with

deterrence factor ϵ *computing some function* f *in the* $\mathcal{L}$ *and* $\mathcal{J}$ *model, if the following security properties are satisfied:*

1. ***Simulatability with*** ϵ***-deterrent****: The protocol* π *(as described above) is secure against a covert adversary according to the strong explicit cheat formulation with* ϵ*-deterrent (see Definition 2) and non-halting detection accurate (see Definition 3).*
2. ***Financial Accountability****: For every PPT adversary* $\mathcal{A}$ *corrupting parties* P_i *for* $i \in \mathcal{I} \subset [n]$*, there exists a negligible function* $\mu(\cdot)$ *such that for all* $(\bar{x}, z) \in (\{0,1\})^{n+1}$ *the following holds:*
   If for any honest party $P_h \in [n] \setminus \mathcal{I}$ *it holds that* $\mathsf{OUTPUT}_h(\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa)) = \mathsf{corrupted}_*$ [3]*, then* $\exists\, m \in \mathcal{I}$ *such that:*

$$\Pr[b_m^{(\mathsf{post})} = b_m^{(\mathsf{pre})} - d] > 1 - \mu(\kappa),$$

   where d *denotes the amount of deposited coins per party.*
3. ***Financial Defamation Freeness****: For every PPT adversary* $\mathcal{A}$ *corrupting parties* P_i *for* $i \in \mathcal{I} \subset [n]$*, there exists a negligible function* $\mu(\cdot)$ *such that for all* $(\bar{x}, z) \in (\{0,1\})^{n+1}$ *and all* $j \in [n] \setminus \mathcal{I}$ *the following holds:*

$$\Pr[b_j^{(\mathsf{post})} < b_j^{(\mathsf{pre})}] < \mu(\kappa).$$

Remark 1: For simplicity, we assume that the adversary does not transfer coins after sending the deposit to $\mathcal{J}$. This assumption can be circumvented by restating financial accountability such that the sum of the balances of all corrupted parties (not just the ones involved in the protocol) is reduced by d.

Remark 2: Similar to the sanity check of [AO12], we note that any maliciously secure protocol can be augmented with a simple $\mathcal{J}$ that only works as a temporary escrow to achieve FBC with deterrence factor $\epsilon = 1 - \mathsf{negl}(\kappa)$. This is easy to see since cheating in a maliciously secure protocol is successful only with negligible probability. Hence, a simple Punish protocol that just transfer back the deposits is enough to satisfy financial accountability and financial defamation freeness.

3.3 Proving Security of Financially Backed Covert Security

Our notion of financially backed covert security (FBC) consists of three properties. The simulatability property requires the protocol π, which augments the covertly secure protocol π', to be covertly secure as well. This does not automatically follows from the security of π', in particular since π includes the broadcast of certificates in case of detected cheating. Showing simulatability of π guarantees that the adversary does not learn sensitive information from the

[3] We use the notation $\mathsf{corrupted}_*$ to denote that the output of P_h is $\mathsf{corrupted}_i$ for some $i \in \mathcal{I}$. We stress that i does not need to be equal to m of the financial accountability property.

certificates. Showing that a protocol π satisfies the simulatability property is proven via a simulation proof. In contrast, we follow a game-based approach to formally prove financial accountability and financial defamation freeness. To this end, we introduce two novel security games, $\mathsf{Exp}^{\mathsf{FA}}$ and $\mathsf{Exp}^{\mathsf{FDF}}$, in the following. Although these two properties are similar to the accountability and defamation freeness properties of PVC, we are the first to introduce formal security games for any of these properties. While we focus on financial accountability and financial defamation freeness, we note that our approach and our security games can be adapted to suit for the security properties of PVC as well.

Both security games are played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. We define the games in a way that allows us to abstract away most of the details of π. In particular, we parameterize the games by two inputs, one for the challenger and one for the adversary. The challenger's input contains the certificates $\{\mathsf{cert}_i\}_{i\in[n]\setminus\mathcal{I}}$ of all honest parties generated by the Blame-algorithm after the execution of π while the adversary's input consists of all malicious parties' views $\{\mathsf{view}_i\}_{i\in\mathcal{I}}$. By introducing the certificates as inputs to the game, we can prove financial accountability and financial defamation freeness independent from proving simulatability of protocol π.

Throughout the execution of the security games, the adversary executes one instance of the punishment protocol Punish with the challenger that takes over the roles of all honest and trusted parties, i.e., the honest protocol parties P_h for $h \notin \mathcal{I}$, the judge $\mathcal{J}$, and the ledger $\mathcal{L}$. To avoid an overly complex challenger description, we define those parties as separated entities that can be addressed by the adversary separately and are all executed by the challenger: $\{\mathtt{P_h}\}_{\mathtt{h}\in[n]\setminus\mathcal{I}}$, $\mathtt{J}$, and $\mathtt{L}$. In case any entity is supposed to act pro-actively and does not only wait to react to malicious behavior, the entity is invoked by the challenger. Communication between said entities is simulated by the challenger. The adversary acts on behalf of the corrupted parties.

Financial accountability game. Intuitively, financial accountability states that whenever any honest party detects cheating, there is some corrupted party that loses her deposit. Therefore, we require that the output of all honest parties was $\mathsf{corrupted}_m$ for $m \in \mathcal{I}$ in the execution of π. If this holds, the security game executes Punish as specified by the FBC protocol. Before the execution of Punish, the challenger asks the ledger for the balances of all parties and stores them as $\{b_i^{(\mathsf{prePunish})}\}_{i\in[n]}$. Note that $\mathsf{prePunish}$ denotes the time before Punish but after the whole protocol already started. This means, relating to Definition 1, the security deposits are already transferred to $\mathcal{J}$, i.e., $b_i^{\mathsf{prePunish}} = b_i^{\mathsf{pre}} - d$. After the execution, the challenger $\mathcal{C}$ again reads the balances of all parties storing them as $\{b_i^{(\mathsf{post})}\}_{i\in[n]}$. If $b_m^{(\mathsf{post})} = b_m^{(\mathsf{prePunish})} + d$ for all $m \in \mathcal{I}$, i.e., all corrupted parties get their deposits back, the adversary wins and the challenger outputs 1, otherwise $\mathcal{C}$ outputs 0. A protocol satisfies the financial accountability property as stated in Definition 1 if for each adversary $\mathcal{A}$ running in time polynomial in κ the probability that $\mathcal{A}$ wins game $\mathsf{Exp}^{\mathsf{FA}}$ is at most negligible, i.e., if $\Pr[\mathsf{Exp}^{\mathsf{FA}}(\mathcal{A}, \kappa) = 1] \leq \mathsf{negl}(\kappa)$.

We provide a graphical description of the game $\mathsf{Exp}^{\mathsf{FA}}$ in Figure 1 of the supplementary materials H.

Financial defamation freeness game. Intuitively, financial defamation freeness states that an honest party can never lose her deposit as a result of executing the Punish protocol. The security game is executed in the same way as the financial accountability game. It only differs in the winning conditions for the adversary. After the execution $\mathcal{C}$ checks the balances of the honest parties. If $b_h^{(\mathsf{post})} < b_h^{(\mathsf{prePunish})} + d$ for at least one $h \in [n] \setminus \mathcal{I}$, the adversary wins and the challenger outputs 1, otherwise $\mathcal{C}$ outputs 0. A protocol satisfies the financial defamation freeness property as stated in Definition 1 if for each adversary $\mathcal{A}$ running in time polynomial in κ the probability that $\mathcal{A}$ wins game $\mathsf{Exp}^{\mathsf{FDF}}$ is at most negligible, i.e. if $\Pr[\mathsf{Exp}^{\mathsf{FDF}}(\mathcal{A}, \kappa) = 1] \leq \mathsf{negl}(\kappa)$.

We provide a graphical description of the game $\mathsf{Exp}^{\mathsf{FDF}}$ in Figure 2 of the supplementary materials H.

4 Features of PVC Protocols

We present transformations from different classes of *publicly verifiable covertly secure* multi-party protocols (PVC) to *financially backed covertly secure* protocols (FBC). As our transformations make use of concrete features of the PVC protocol (e.g., the exchanged messages), we cannot use the PVC protocol in a block-box way. Instead, we model the PVC protocol in an abstract way, stating features that are required by our constructions. In the remainder of this section, we present these features in detail and describe how we model them. We note that all existing PVC multi-party protocols [DOS20, FHKS21, SSS21] provide the features specified in this section.

4.1 Cut-and-Choose

Although not required per definition of PVC, a fundamental technique used by all existing PVC protocols is the *cut-and-choose* approach that leverages a semi-honest protocol by executing t instances of the semi-honest protocol in parallel. Afterwards, the views (i.e., input and randomness) of the parties is revealed in s instances. This enables parties to detect misbehavior with probability $\epsilon = \frac{s}{t}$. PVC protocols can be split into protocols where parties provide private inputs and those where parties do not have secret data. While cut-and-choose for input-independent protocols, i.e., those where parties do not have private inputs, work as explained on a high level before, the approach must be utilized in such a way that input privacy is guaranteed for input-dependent protocols. However, for both classes of protocols, a cheat detection probability of $\epsilon = \frac{s}{t}$ can be achieved. We elaborate more the two variants and provide details about them in the supplementary materials C.

4.2 Verification of Protocol Executions

An important feature of PVC protocols based on cut-and-choose is to enable parties to verify the execution of the opened protocol instances. This requires parties to emulate the protocol messages and compare them with the messages exchanged during the real execution. In order to emulate honest behavior, we need the protocol to be derandomized.

Derandomization of the protocol execution. In general, the behavior of each party during some protocol execution depends on the party's private input, its random tape and all incoming messages. In order to enable parties to check the behavior of other parties in retrospect, the actions of all parties need to be made deterministic. To this end, we require the feature of a PVC protocol that all random choices of a party P_i in a protocol instance are derived from some random seed seed_i using a pseudorandom generator (PRG). The seed seed_i is fixed before the beginning of the execution. It follows that the generated outgoing messages are computed deterministically given the seed seed_i, the secret input and all incoming messages.

State evolution. Corresponding to our communication model (cf. Section 2), the internal states of the parties in a semi-honest protocol instance evolve in rounds. For each party P_i, for $i \in [n]$, and each round $k > 0$ the protocol defines a state transition $\mathsf{computeRound}_k^i$ that on input the previous internal state $\mathsf{state}_{(k-1)}^{(i)}$ and the set of incoming messages $\{\mathsf{msg}_{(k-1)}^{(j,i)}\}_{j\neq i}$ computes the new internal state $\mathsf{state}_{(k)}^{(i)}$ and the set of outgoing messages $\{\mathsf{msg}_{(k)}^{(i,j)}\}_{j\neq i}$. Based on the derandomization feature, the state transition is deterministic, i.e., all random choices are derived from a random seed included in the internal state of a party. Each party starts with an initial internal state that equals its random seed seed_i and its secret input x_i. In case no secret input is present (i.e., in the input-independent setting) or no message is sent, the value is considered to be a dummy symbol ($\perp$). We denote the set of all messages sent during a protocol instance by *protocol transcript*. Summarizing, we formally define

$$\begin{aligned}
\mathsf{state}_{(0)}^{(i)} &\leftarrow (\mathsf{seed}_i, x_i)\\
\{\mathsf{msg}_{(0)}^{(j,i)}\}_{j\in[n]\setminus\{i\}} &\leftarrow \{\perp\}_{j\in[n]\setminus\{i\}}\\
(\mathsf{state}_{(k)}^{(i)}, \{\mathsf{msg}_{(k)}^{(i,j)}\}_{j\in[n]\setminus\{i\}}) &\leftarrow \mathsf{computeRound}_k^i(\mathsf{state}_{(k-1)}^{(i)}, \{\mathsf{msg}_{(k-1)}^{(j,i)}\}_{j\in[n]\setminus\{i\}}).
\end{aligned}$$

Protocol emulation. In order to check for malicious behavior, parties locally emulate the protocol execution of the opened instances and compare the set of computed messages with the received ones. In case some involved parties are not checked (e.g., in the input-dependent setting), the emulation gets their messages as input and assumes them to be correct. In this case, in order to ensure that each party can run the emulation, it is necessary that each party has access to all messages sent in the opened instance (cf. Section 4.4).

To formalize the protocol emulation, we define for each n-party protocol π with R rounds two emulation algorithms. The first algorithm $\mathsf{emulate}_{\pi}^{\mathsf{full}}$ emulates all parties while the second algorithm $\mathsf{emulate}_{\pi}^{\mathsf{part}}$ emulates only a partial subset of the parties and considers the messages of all other parties as correct. We formally define them as

$$(\{\mathsf{msg}_{(k)}^{(i,j)}\}_{k,i,j\neq i}, \{\mathsf{state}_{(k)}^{(i)}\}_{k,i}) \leftarrow \mathsf{emulate}_{\pi}^{\mathsf{full}}(\{\mathsf{state}_{(0)}^{(i)}\}_{i}) \quad \text{and}$$
$$(\{\mathsf{msg}_{(k)}^{(i,j)}\}_{k,i,j\neq i}, \{\mathsf{state}_{(k)}^{(\hat{i})}\}_{k,\hat{i}}) \leftarrow \mathsf{emulate}_{\pi}^{\mathsf{part}}(O, \{\mathsf{state}_{(0)}^{(\hat{i})}\}_{\hat{i}}, \{\mathsf{msg}_{(k)}^{(i^*,j)}\}_{k,i^*,j\neq i^*})$$

where $k \in [R]$, $i, j \in [n]$, $\hat{i} \in O$ and $i^* \in [n] \setminus O$. O denotes the set of opened parties.

4.3 Deriving the Initial States

As a third feature, we require a mechanism for the parties of a PVC protocol to learn the initial states of all opened parties in order to perform the protocol emulation (cf. Section 4.2). Since PVC prevents detection dependent abort, parties learn the initial state even if the adversary aborts after having learned the cut-and-choose selection. Existing multi-party PVC protocols provide this feature by either making use of oblivious transfer or time-lock puzzles as in [DOS20] resp. [FHKS21, SSS21]. We elaborate on these protocols in the supplementary materials C.

To model this behavior formally, we define the abstract tuples $\mathsf{initData}^{\mathsf{core}}$ and $\mathsf{initData}^{\mathsf{aux}}$ as well as the algorithm $\mathsf{deriveInit}$. $\mathsf{initData}_{(i)}^{\mathsf{core}}$ represents data each party holds that should be signed by P_i and can be used to derive the initial state of party P_i in a single protocol instance (e.g., a signed time-lock puzzle). $\mathsf{initData}_{(i)}^{\mathsf{aux}}$ represents the additional data all parties receive during the PVC protocol that can be used to interpret $\mathsf{initData}_{(i)}^{\mathsf{core}}$ (e.g., the verifiable solution of the time-lock puzzle). Finally, $\mathsf{deriveInit}$ is an algorithm that on input $\mathsf{initData}_{(i)}^{\mathsf{core}}$ and $\mathsf{initData}_{(i)}^{\mathsf{aux}}$ derives the initial state of party P_i (e.g., verifying the solution of the puzzle). Instead of outputting an initial state, the algorithm $\mathsf{deriveInit}$ can also output bad or $\perp$. The former states that party P_i misbehaved during the PVC protocol by providing inconsistent data. The symbol $\perp$ states that the input to $\mathsf{deriveInit}$ has been invalid which can only occur if $\mathsf{initData}_{(i)}^{\mathsf{core}}$ or $\mathsf{initData}_{(i)}^{\mathsf{aux}}$ have been manipulated.

Similar to commitment schemes, our abstraction satisfies a *binding* and *hiding* requirement, i.e., it is computationally *binding* and computationally *hiding*. The binding property requires that the probability of any polynomial time adversary finding a tuple (x, y_1, y_2) such that $\mathsf{deriveInit}(x, y_1) \neq \perp$, $\mathsf{deriveInit}(x, y_2) \neq \perp$, and $\mathsf{deriveInit}(x, y_1) \neq \mathsf{deriveInit}(x, y_2)$ is negligible. The hiding property requires that the probability of a polynomial time adversary finding for a given $\mathsf{initData}^{\mathsf{core}}$ a $\mathsf{initData}^{\mathsf{aux}}$ such that $\mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}, \mathsf{initData}^{\mathsf{aux}}) \neq \perp$ is negligible.

4.4 Public Transcript

A final feature required by PVC protocols of class 1 and 2 is the availability of a common public transcript. We define three levels of transcript availability. First, a *common public transcript of messages* ensures that all parties hold a common transcript containing all messages that have been sent during the execution of a protocol instance. Every protocol can be transformed to provide this feature by requiring all parties to send all messages to all other parties and defining a fixed ordering on the sent messages – we consider an ordering of messages by the round they are sent, the index of the sender, and the receiver's index in this sequence. If messages should be secret, each pair of parties executes a secure key exchange as part of the protocol instance and then encrypts messages with the established keys. Agreement is achieved by broadcasting signatures on the transcript, e.g., via signing the root of a Merkle tree over all message hashes as discussed in [FHKS21] and required in our transformations. Second, a *common public transcript of hashes* ensures that all parties hold a common transcript containing the hashes of all messages sent during the execution of a protocol instance. This feature is achieved similar to the transcript of messages but parties only send message hashes to all parties that are not the intended receiver. Finally, the *private transcript* does not require any agreement on the transcript of a protocol instance.

Currently, all existing multi-party PVC protocols either provide a common public transcript of messages [DOS20, FHKS21] or a common public transcript of hashes [SSS21]. However, [DOS20] and [FHKS21] can be trivially adapted to provide just a common public transcript of hashes.

5 Building Blocks

In this section, we describe the building blocks for our financially backed covertly secure protocols. In the supplementary materials D, we show security of the building blocks and that incorporating the building blocks into the PVC protocol does not affect the protocol's security.

5.1 Internal State Commitments

To realize the judge in an efficient way, we want it to validate just a single protocol step instead of validating a whole instance. Existing PVC protocols prove misbehavior in a naive way by allowing parties to show that some other party P_j had an initial state $\mathsf{state}_{(0)}^{(j)}$. Based on the initial state, the judge recomputes the whole protocol instance. In contrast to this, we incorporate a mechanism that allows parties to prove that P_j has been in state $\mathsf{state}_{(k)}^{(j)}$ in a specific round k where misbehavior was detected. Then, the judge just needs to recompute a single step. To this end, we require that parties commit to each intermediate internal state during the execution of each semi-honest instance in a publicly verifiable way. In particular, in each round k of each semi-honest instance ℓ,

each party P_i sends a hash of its internal state to all other parties using a collision-resistant hash function $H(\cdot)$, i.e., $H(\mathsf{state}^{(i)}_{(\ell,k)})$. At the end of a protocol instance each party P_h creates a Merkle tree over all state hashes, i.e., $\mathsf{sTree}_\ell := \mathsf{MTree}(\{\mathsf{hash}^{(i)}_{(\ell,k)}\}_{k\in[R],i\in[n]})$, and broadcasts a signature on the root of this tree, i.e., $\left\langle \mathsf{MRoot}(\mathsf{sTree}_\ell)\right\rangle_h$.

5.2 Signature Encoding

Our protocol incorporates signatures in order to provide evidence to the judge $\mathcal{J}$ about the behavior of the parties. Without further countermeasures, an adversary can make use of signed data across multiple instances or rounds, e.g., she could claim that some message msg sent in round k has been sent in round k' using the signature received in round k. To prevent such an attack, we encode signed data by prefixing it with the corresponding indices before being signed. Merkle tree roots are prefixed with the instance index ℓ. Message hashes are prefixed with ℓ, the round index k, the sender index i and the receiver index j. Initial state commitments ($\mathsf{initData}^{\mathsf{core}}_{(\ell,i)}$) are prefixed with ℓ and the index i of the party who's initial state the commitment refers to. The signature verification algorithm automatically checks for correct prefixing. The indices are derived from the super- and subscripts. If one index is not explicitly provided, e.g., in case only one instance is executed, the index is assumed to be 1.

5.3 Bisection of Trees

Our constructions make heavily use of Merkle trees to represent sets of data. This enables parties to efficiently prove that chunk of data is part of a set by providing a Merkle proof showing that the chunk is a leaf of the corresponding Merkle tree. In case two parties disagree about the data of a Merkle tree which should be identical, we use a bisection protocol Π_{BS} to narrow down the dispute to the first leaf of the tree on which they disagree. This helps a judging party to determine the lying party by just verifying a single data chunk in contrast to checking the whole data. The technique of bisecting was first used by Canetti et al. [CRR11] in the context of verifiable computing. Later, the technique was used in [KGC+18, TR19, EFS20].

The protocol is executed between a party P_b with input a tree mTree_b, a party P_m with input a tree mTree_m and a trusted judge $\mathcal{J}$ announcing three public inputs: root^j, the root of mTree_j as claimed by P_j for $j \in \{b, m\}$, and width, the width of the trees, i.e., the number of leaves. The protocol returns the index z of the first leaf at which mTree_b and mTree_m differentiate, the leaf hash^m_z at position z of mTree_m, and the common leaf $\mathsf{hash}_{(z-1)}$ at position $z-1$. The latter is $\perp$ if $z = 1$. Let $\mathsf{node}(\mathsf{mTree}, x, y)$ be the node of a tree mTree at position x of layer y – positions start with 1. The protocol is executed as follows:

Protocol Bisection Π_{BS}

1. $\mathcal{J}$ initializes layer variable $y := 1$, position variable $x := 1$, last agreed hash $\mathsf{hash}^a := \perp$, and $\mathsf{depth} := \lceil \log_2(\mathsf{width}) \rceil + 1$
2. All parties repeat this step while $y \leq \mathsf{depth}$:
   (a) Both P_j (for $j \in \{b, m\}$) send $\mathsf{hash}^j := \mathsf{node}(\mathsf{mTree}_j, x, y)$ and $\sigma^j := \mathsf{MProof}(\mathsf{hash}^j, \mathsf{mTree}_j)$ to $\mathcal{J}$.
   (b) If $\mathsf{MVerify}(\mathsf{hash}^j, x, \mathsf{root}^j, \sigma^j) = \mathsf{false}$ (for $j \in \{b, m\}$), $\mathcal{J}$ discards the message from P_j.
   (c) If $y = \mathsf{depth}$, $\mathcal{J}$ keeps hash^b and hash^m and sets $y = y + 1$.
   (d) If $y < \mathsf{depth}$ and $\mathsf{hash}^b = \mathsf{hash}^m$, $\mathcal{J}$ sets $x = (2 \cdot x) + 1$ and $y = y + 1$.
   (e) If $y < \mathsf{depth}$ and $\mathsf{hash}^b \neq \mathsf{hash}^m$, $\mathcal{J}$ sets $x = (2 \cdot x) - 1$ and $y = y + 1$.
3. If $\mathsf{hash}^b = \mathsf{hash}^m$
   – $\mathcal{J}$ sets $z := x + 1$ and $\mathsf{hash}_{(z-1)} := \mathsf{hash}_b$.
   – P_m sends $\mathsf{hash}_z^m := \mathsf{node}(\mathsf{mTree}_m, z, \mathsf{depth})$ and $\sigma := \mathsf{MProof}(\mathsf{hash}_z^m, \mathsf{mTree}_m)$ to $\mathcal{J}$.
   – If $\mathsf{MVerify}(\mathsf{hash}_z^m, z, \mathsf{root}, \sigma) = \mathsf{false}$, $\mathcal{J}$ discards. Otherwise $\mathcal{J}$ stores hash_z^m.
4. If $\mathsf{hash}^b \neq \mathsf{hash}^m$
   – $\mathcal{J}$ sets $z := x$ and $\mathsf{hash}_z^m := \mathsf{hash}^m$. If $z = 1$, $\mathcal{J}$ sets $\mathsf{hash}_{(z-1)} := \perp$, and the protocol jumps to step 5.
   – P_m sends $\mathsf{hash}_{(z-1)} := \mathsf{node}(\mathsf{mTree}_m, z - 1, \mathsf{depth})$ and $\sigma := \mathsf{MProof}(\mathsf{hash}_{(z-1)}, \mathsf{mTree}_m)$ to $\mathcal{J}$.
   – If $\mathsf{MVerify}(\mathsf{hash}_{(z-1)}, z - 1, \mathsf{mTree}_m, \sigma) = \mathsf{false}$, $\mathcal{J}$ discards. Otherwise, $\mathcal{J}$ keeps $\mathsf{hash}_{(z-1)}$.
5. $\mathcal{J}$ announces public outputs z, hash_z^m and $\mathsf{hash}_{(z-1)}$.

6 Class 1: Input-Independent with Public Transcript

Our first transformation builds on input-independent PVC protocols where all parties possess a common public transcript of hashes (cf. Section 4.4) for each checked instance. Since the parties provide no input in these protocols, all parties can be opened. The set of input-independent protocols includes the important class of preprocessing protocols. In order to speed up MPC protocols, a common approach is to split the computation in an *offline* and an *online* phase. During the offline phase, precomputations are carried out to set up some correlated randomness. This phase does not require the actual inputs and can be executed continuously. In contrast, the online phase requires the private inputs of the parties and consumes the correlated randomness generated during the offline phase to speed up the execution. As the online performance is more time critical, the goal is to put as much work as possible into the offline phase. Prominent examples following this approach are the protocols of Damgård et al. [DPSZ12, DKL+13] and Wang et al. [WRK17a, WRK17b, YWZ20]. Input-independent PVC protocols with a public transcript can be obtained from semi-honest protocols using the input-independent compilers of Damgård et al.[DOS20] and Faust et al. [FHKS21].

In order to apply our construction to an input-independent PVC protocol, π^{pp}, we require π^{pp} to provide some features presented in Section 4 and to have

incorporated some of the building blocks described in Section 5. First, we require the PVC protocol to be based on the cut-and-choose approach (cf. Section 4.1). Second, we require the actions of each party P_i in a protocol execution to be deterministically determined by a random seed (cf. Section 4.2). Third, we require that all parties learn the initial states of all other parties in the opened protocol instances (cf. Section 4.3). To this end, the parties receive signed data (e.g., a commitment and decommitment value) to derive the initial states of the other parties. Fourth, parties need to commit to their intermediate internal states during the protocol executions in a publicly verifiable way (cf. Section 5.1). Finally, all signed data match the encoded form specified in Section 5.2.

In order to achieve the public transcript of hashes and the commitments to the intermediate internal states, parties exchange additional data in each round. Formally, whenever some party P_h in round k of protocol instance ℓ transitions to a state $\mathsf{state}^{(h)}_{(\ell,k)}$ with the outgoing messages $\{\mathsf{msg}^{(h,i)}_{(\ell,k)}\}_{i\in[n]\setminus\{h\}}$, then it actually sends the following to P_i:

$$(\mathsf{msg}^{(h,i)}_{(\ell,k)}, \{\mathsf{hash}^{(h,j)}_{(\ell,k)} := H(\mathsf{msg}^{(h,j)}_{(\ell,k)})\}_{j\in[n]\setminus\{h,i\}}, \mathsf{hash}^{(h)}_{(\ell,k)} := H(\mathsf{state}^{(h)}_{(\ell,k)}))$$

Let O denote the set of opened instances. We summarize the aforementioned requirements by specifying the data that the view of any honest party P_h includes. It contains signed data to derive the initial state of all parties for the opened instances, i.e.,

$$\{(\langle \mathsf{initData}^{\mathsf{core}}_{(i,\ell)} \rangle_i, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})\}_{\ell\in O, i\in[n]}, \tag{1a}$$

a Merkle tree over the hashes of all messages exchanged within a single instance for all instances, i.e.,

$$\{\mathsf{mTree}_\ell\}_{\ell\in[t]} := \{\mathsf{MTree}(\{\mathsf{hash}^{(i,j)}_{(\ell,k)}\}_{k\in[R],i\in[n],j\neq i})\}_{\ell\in[t]}, \tag{1b}$$

a Merkle tree over the hashes of all intermediate internal states of a single instance for all instances, i.e,

$$\{\mathsf{sTree}_\ell\}_{\ell\in[t]} := \{\mathsf{MTree}(\{\mathsf{hash}^{(i)}_{(\ell,k)}\}_{k\in[R],i\in[n]})\}_{\ell\in[t]} \tag{1c}$$

and signatures from each party over the roots of the message and state trees, i.e,

$$\{\langle \mathsf{MRoot}(\mathsf{mTree}_\ell)\rangle_i\}_{i\in[n],\ell\in[t]} \tag{1d}$$

and

$$\{\langle \mathsf{MRoot}(\mathsf{sTree}_\ell)\rangle_i\}_{i\in[n],\ell\in[t]}. \tag{1e}$$

We next define the blame algorithm that takes the specified view as input and continue with the description of the punishment protocol afterwards.

The blame algorithm. At the end of protocol π^{pp}, all parties execute the blame algorithm $\mathsf{Blame}^{\mathsf{pp}}$ to generate a certificate cert. The resulting certificate is broadcasted and the honest party finishes the execution of π^{pp} by outputting cert. The certificate is generated as follows:

Algorithm $\mathsf{Blame}^{\mathsf{pp}}$

1. P_h runs $\mathsf{state}_{(\ell,0)}^{(i)} = \mathsf{deriveInit}(\mathsf{initData}_{(i,\ell)}^{\mathsf{core}}, \mathsf{initData}_{(i,\ell)}^{\mathsf{aux}})$ for each $i \in [n], \ell \in O$. Let $\mathcal{B}$ be the set of all tuples $(\ell, 0, m, 0)$ such that $\mathsf{state}_{(\ell,0)}^{(m)} = \mathsf{bad}$. If $\mathcal{B} \neq \emptyset$, goto step 4.
2. P_h emulates for each $\ell \in O$ the protocol executions on input the initial states from all parties to obtain the expected messages and the expected intermediate states of all parties, i.e., $(\{\mathsf{msg}_{(\ell,k)}^{(i,j)}\}_{k\in[R],i\in[n],j\neq i}, \{\mathsf{state}_{(\ell,k)}^{(i)}\}_{k,i,j}) := \mathsf{emulate}^{\mathsf{full}}(\{\mathsf{state}_{(\ell,0)}^{(i)}\}_{i\in[n]})$.
3. Let $\mathcal{B}$ be the set of all tuples (ℓ, k, m, i) such that $H(\mathsf{msg}_{(\ell,k)}^{(m,i)}) \neq \mathsf{hash}_{(\ell,k)}^{(m,i)}$ or $H(\mathsf{state}_{(\ell,k)}^{(m)}) \neq \mathsf{hash}_{(\ell,k)}^{(m)}$ – where $\mathsf{hash}_{(\ell,k)}^{(m,i)}$ and $\mathsf{hash}_{(\ell,k)}^{(m)}$ are extracted from mTree_ℓ or sTree_ℓ respectively. In case of an incorrect state hash, set $i = 0$.
4. If $\mathcal{B} = \emptyset$ P_h outputs $\mathsf{cert} := \bot$. Otherwise, P_h picks the tuple (ℓ, k, m, i) from $\mathcal{B}$ with the smallest ℓ, k, m, i in this sequence, sets $k' := k - 1$ and defines variables as follows – variables that are not explicitly defined are set to $\bot$.

$$
\begin{aligned}
\text{(Always):} \qquad ids &:= (\ell, k, m, i)\\
\mathsf{initData} &:= (\langle \mathsf{initData}_{(\ell,m)}^{\mathsf{core}} \rangle_m, \mathsf{initData}_{(\ell,m)}^{\mathsf{aux}})\\
\mathsf{root}^{\mathsf{state}} &:= \langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_m\\
\mathsf{root}^{\mathsf{msg}} &:= \langle \mathsf{MRoot}(\mathsf{mTree}_\ell) \rangle_m\\
\text{(If } k > 0\text{):} \quad \mathsf{state}_{out} &:= (\mathsf{hash}_{(\ell,k)}^{(m)}, \mathsf{MProof}(\mathsf{hash}_{(\ell,k)}^{(m)}, \mathsf{sTree}_\ell))\\
\mathsf{msg}_{out} &:= (\mathsf{hash}_{(\ell,k)}^{(m,i)}, \mathsf{MProof}(\mathsf{hash}_{(\ell,k)}^{(m,i)}, \mathsf{mTree}_\ell))\\
\text{(If } k > 1\text{):} \quad \mathsf{state}_{in} &:= (\mathsf{state}_{(\ell,k')}^{(m)}, \mathsf{MProof}(H(\mathsf{state}_{(\ell,k')}^{(m)}), \mathsf{sTree}_\ell))\\
\mathcal{M}_{in} &:= \{(\mathsf{msg}_{(\ell,k')}^{(j,m)}, \mathsf{MProof}(H(\mathsf{msg}_{(\ell,k')}^{(j,m)}), \mathsf{mTree}_\ell))\}_{j\in[n]}
\end{aligned}
$$

5. Output $\mathsf{cert} := (ids, \mathsf{initData}, \mathsf{root}^{\mathsf{state}}, \mathsf{root}^{\mathsf{msg}}, \mathsf{state}_{in}, \mathcal{M}_{in}, \mathsf{state}_{out}, \mathsf{msg}_{out})$.

The punishment protocol. Each party P_i (for $i \in [n]$) checks if $\mathsf{cert} \neq \bot$. If this is the case, P_i sends cert to $\mathcal{J}^{\mathsf{pp}}$ [4]. Otherwise, P_i waits till time T to receive her deposit back. Timeout T is set such that the parties have sufficient time to submit a certificate after the execution of π^{pp} and $\mathsf{Blame}^{\mathsf{pp}}$. The judge $\mathcal{J}^{\mathsf{pp}}$ is described in the following. For the sake of compactness, we extracted the description of the validation algorithms $\mathsf{wrongMsg}$ and $\mathsf{wrongState}$ and the algorithm $\mathsf{getIndex}$ into Section J of the supplementary material. We stress that the validation algorithms $\mathsf{wrongMsg}$ and $\mathsf{wrongState}$ don't need to recompute a whole protocol execution but only a single step. Therefore, $\mathcal{J}^{\mathsf{pp}}$ is very efficient and can, for instance, be realized via a smart contract. To be more precise, the judge is execution without any interaction and runs in computation complexity linear in the protocol complexity. By allowing logarithmic interactions between the judge and the parties, we can further reduce the computation complexity

[4] A practical implementation would incorporate a mechanism that avoids duplicated submissions.

to logarithmic in the protocol complexity. This can be achieved by applying the efficiency improvement described in supplementary materials G.

Judge $\mathcal{J}^{\mathsf{pp}}$

Initialization: The judge has access to public variables n, t, T and the set of parties $\{P_i\}_{i\in[n]}$. Further, it maintains a set $\mathsf{cheaters}$ initially set to $\emptyset$. Prior to the execution of π^{pp}, $\mathcal{J}^{\mathsf{pp}}$ has received d coins from each party P_i.

Proof verification: Wait until time T_1 to receive $((\ell,k,m,i),\mathsf{initData},\langle\mathsf{root}^{\mathsf{state}}_{(\ell)}\rangle_m,\langle\mathsf{root}^{\mathsf{msg}}_{(\ell)}\rangle_m,\mathsf{state}_{in},\mathcal{M}_{in},\mathsf{state}_{out},(\mathsf{hash},\sigma))$ and do:

1. If $P_m \in \mathsf{cheaters}$, abort.
2. Parse $\mathsf{initData}$ to $(\langle\mathsf{initData}^{\mathsf{core}}_{(\ell,m)}\rangle_m,\mathsf{initData}^{\mathsf{aux}}_{(\ell,m)})$ and set $\mathsf{state}_0 = \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(\ell,m)},\mathsf{initData}^{\mathsf{aux}}_{(\ell,m)})$. If $\mathsf{Verify}(\langle\mathsf{initData}^{\mathsf{core}}_{(\ell,m)}\rangle_m) = \mathsf{false}$ or $\mathsf{state}_0 = \bot$, abort. If $\mathsf{state}_0 = \mathsf{bad}$, add P_m to $\mathsf{cheaters}$ and stop.
3. If $\mathsf{Verify}(\langle\mathsf{root}^{\mathsf{state}}_{(\ell)}\rangle_m) = \mathsf{false}$ or $\mathsf{Verify}(\langle\mathsf{root}^{\mathsf{msg}}_{(\ell)}\rangle_m) = \mathsf{false}$, abort.
4. If $i = 0$ and $\mathsf{wrongState}(\mathsf{state}_0,\mathsf{state}_{in},\mathsf{state}_{out},\mathcal{M}_{in},\mathsf{root}^{\mathsf{state}}_{(\ell)},\mathsf{root}^{\mathsf{msg}}_{(\ell)},\ell,k,m) = \mathsf{true}$, add P_m to $\mathsf{cheaters}$.
5. If $i > 0$, $\mathsf{MVerify}(\mathsf{hash},\mathsf{getIndex}(k,m,i),\mathsf{root}^{\mathsf{msg}}_{(\ell)},\sigma) = \mathsf{true}$ and $\mathsf{wrongMsg}(\mathsf{state}_0,\mathsf{state}_{in},\mathsf{hash},\mathcal{M}_{in},,\mathsf{root}^{\mathsf{state}}_{(\ell)},\mathsf{root}^{\mathsf{msg}}_{(\ell)},\ell,m,k,i) = \mathsf{true}$, add P_m to $\mathsf{cheaters}$.

Timeout: At time T_1, send d coins to each party $P_i \notin \mathsf{cheaters}$.

6.1 Security

Theorem 1. *Let $(\pi^{\mathsf{pp}},\cdot,\cdot)$ be an n-party publicly verifiable covert protocol computing function f with deterrence factor ϵ satisfying the view requirements stated in Eq.* (1a)-(1e). *Further, let the signature scheme* $(\mathsf{Generate},\mathsf{Sign},\mathsf{Verify})$ *be existentially unforgeable under chosen-message attacks, the Merkle tree satisfies the binding property and the hash function H be collision resistant. Then the protocol π^{pp} together with algorithm $\mathsf{Blame}^{\mathsf{pp}}$, protocol $\mathsf{Punish}^{\mathsf{pp}}$ and judge $\mathcal{J}^{\mathsf{pp}}$ satisfies financially backed covert security with deterrence factor ϵ according to Definition 1.*

We formally prove Theorem 1 in the supplementary materials E.

7 Class 3: Input-Independent with Private Transcript

At the time of writing, there exists no PVC protocol without public transcript that could be directly transformed into an FBC protocol. Moreover, it is not clear, if it is possible to construct a PVC protocol without a public transcript. Instead, we present a transformation from an input-independent PVC protocol with public transcript into an FBC protocol without any form of common public transcript. As in our first transformation, we start with an input-independent PVC protocol π_3^{pvc} that is based on cut-and-choose where parties share a common public transcript. Due to the input-independence, all parties of the checked

instances can be opened. However, unlike our first transformation, which utilizes the public transcript, we remove this feature from the PVC protocol as part of the transformation. We denote the protocol that results by removing the public transcript feature from π_3^{pvc} by π_3. Without having a public transcript, the punishment protocol becomes interactive and more complicated. Intuitively, without a public transcript it is impossible to immediately decide if a message that deviates from the emulation is maliciously generated or is invalid because of a received invalid messages. Note that we still have a common public tree of internal state hashes in our exposition. However, the necessity of this tree can also be removed by applying the techniques presented here that allow us to remove the common transcript.

In order to apply our construction to a protocol π_3, we require almost the same features of π_3 as demanded in our first transformation (cf. Section 6). For the sake of exposition, we outline the required features here again and point out the differences. First, we require π_3 to be based on the cut-and-choose approach (cf. Section 4.1). Second, we require the actions of each party P_i in a semi-honest instance execution to be deterministically determined by a random seed (cf. Section 4.2). Third, we require that all parties learn the initial states of all other parties in the opened protocol instances (cf. Section 4.3). To this end, the parties receive signed data (e.g., a commitment and decommitment value) to derive the initial states of the other parties. Fourth, parties need to commit to their intermediate internal states during the protocol executions in a publicly verifiable way (cf. Section 5.1). Finally, all signed data match the encoded form specified in Section 5.2.

In contrast to the transformation in Section 6 we no longer require from protocol π_3 that the parties send all messages or message hashes to all other parties. Formally, whenever some party P_h in round k of protocol instance ℓ transitions to a state $\mathsf{state}_{(\ell,k)}^{(h)}$ with the outgoing messages $\{\mathsf{msg}_{(\ell,k)}^{(h,i)}\}_{i\in[n]\setminus\{h\}}$, then it actually sends the following to P_i:

$$(\langle \mathsf{msg}_{(\ell,k)}^{(h,i)} \rangle_h, \mathsf{hash}_{(\ell,k)}^{(h)} := H(\mathsf{state}_{(\ell,k)}^{(h)}))$$

Let O be the set of opened instances. We summarize the aforementioned requirements by specifying the data that the view of any honest party P_h after the execution of π_3 includes. The view contains data to derive the initial state of all parties which is signed by each party for each party and every opened instance, i.e.,

$$\{(\langle \mathsf{initData}_{(i,\ell)}^{\mathsf{core}} \rangle_j, \mathsf{initData}_{(i,\ell)}^{\mathsf{aux}})\}_{\ell\in O, i\in[n], j\in[n]}, \tag{2a}$$

a Merkle tree over the hashes of all intermediate internal states of a single instance for all instances, i.e.,

$$\{\mathsf{sTree}_\ell\}_{\ell\in[t]} := \{\mathsf{MTree}(\{\mathsf{hash}_{(\ell,k)}^{(i)}\}_{k\in[R],i\in[n]})\}_{\ell\in[t]}, \tag{2b}$$

signatures from each party over the roots of the state trees, i.e.,

$$\{\langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_i\}_{i\in[n],\ell\in[t]} \tag{2c}$$

and the signed incoming message, i.e.,

$$\mathcal{M} := \{\langle \mathsf{msg}^{(i,h)}_{(\ell,k)} \rangle_i\}_{\ell\in[t],k\in[R],i\in[n]\setminus\{h\}}. \tag{2d}$$

The blame algorithm. At the end of protocol π_3, all parties first execute an evidence algorithm $\mathsf{Evidence}$ to generate partial certificates cert'. The partial certificate is a candidate to be used for the punishment protocol and is broadcasted to all other parties as part of π_3. In case the honest party detects cheating in several occurrences, the party picks the occurrence with the smallest indices (ℓ, k, m, i) (in this sequence). The algorithm to generate partial certificates $\mathsf{Evidence}$ is formally described as follows:

Algorithm $\mathsf{Evidence}$

1. P_h runs $\mathsf{state}^{(i)}_{(\ell,0)} = \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(i,\ell)}, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})$ for each $i \in [n], \ell \in O$. Let $\mathcal{B}$ be the set of all tuples $(\ell, 0, m, 0)$ such that $\mathsf{state}^{(m)}_{(\ell,0)} = \mathsf{bad}$. If $\mathcal{B} \neq \emptyset$, goto step 4.
2. P_h emulates for each $\ell \in O$ the protocol executions on input the initial states from all parties to obtain the expected messages and the expected intermediate states of all parties, i.e., $(\{\widetilde{\mathsf{msg}}^{(i,j)}_{(\ell,k)}\}_{k\in[R],i\in[n],j\neq i}, \{\mathsf{state}^{(i)}_{(\ell,k)}\}_{k,i,j}) := \mathsf{emulate}^{\mathsf{full}}(\{\mathsf{state}^{(i)}_{(\ell,0)}\}_{i\in[n]})$.
3. Let $\mathcal{B}$ be the set of all tuples (ℓ, k, m, h) such that $\mathsf{msg}^{(m,h)}_{(\ell,k)} \neq \widetilde{\mathsf{msg}}^{(m,h)}_{(\ell,k)}$ or $H(\mathsf{state}^{(m)}_{(\ell,k)}) \neq \mathsf{hash}^{(m)}_{(\ell,k)}$ – where $\mathsf{msg}^{(m,h)}_{(\ell,k)}$ and $\mathsf{hash}^{(m)}_{(\ell,k)}$ are taken from $\mathcal{M}$ or sTree_ℓ respectively. In case of an invalid state, set $h = 0$.
4. Pick the tuple (ℓ, k, m, i) from $\mathcal{B}$ with the smallest ℓ, k, m, i in this sequence. If $k > 0$ set $\mathsf{msg}_{out} := \langle \mathsf{msg}^{(m,i)}_{(\ell,k)} \rangle_m$. Otherwise, set $\mathsf{msg}_{out} := \bot$.
5. Output partial certificate $(ids, \mathsf{msg}_{out})$.

Since π_3 does not contain a public transcript of messages, parties can only validate their own incoming message instead of all messages as done in previous approaches. Hence, it can happen that different honest parties generate and broadcast different partial certificates. Therefore, all parties validate the incoming certificates, discard invalid ones and pick the partial certificate cert' with the smallest indices (ℓ, k, m, i) (in this sequence) as their own. If no partial certificate has been received or created, parties set $\mathsf{cert}' := \bot$.

Finally, each honest party executes the blame algorithm $\mathsf{Blame}^{\mathsf{sp}}$ to create the full certificate that is used for both, blaming a malicious party and defending against incorrect accusations. As in this scenario the punishment protocol requires input of accused honest parties, the blame algorithm returns a certificate even if no malicious behavior has been detected, i.e., if $\mathsf{cert}' = \bot$. The final certificate is generated by appending following data from the view to the certificate: $\{(\langle \mathsf{initData}^{\mathsf{core}}_{(i,\ell)} \rangle_j, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})\}_{\ell\in O,i\in[n],j\in[n]}$ (cf. Eq 2a), $\{\mathsf{sTree}_\ell\}_{\ell\in[t]}$ (cf. Eq 2b), and $\{\langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_i\}_{i\in[n],\ell\in[t]}$ (cf. Eq 2c). All the appended data is public and does not really need to be broadcasted. However, in order to match the formal specification, all parties broadcast their whole certificate. If $\mathsf{cert}' \neq \bot$, the honest party outputs in addition to the certificate $\mathsf{corrupted}_m$.

To ease the specification of the punishment protocol in which parties derive further data from the certificates, we define an additional algorithm $\mathsf{mesHistory}$ that uses the messages obtained during the emulation ($\widetilde{\mathsf{msg}}$)[5] to compute the message history up to a specific round k' (inclusively) of instance ℓ. We structure the message history in two layers. For each round $k^* < k'$, parties create a Merkle tree of all messages emulated in this round. These trees constitute the bottom layer. On the top layer, parties create a Merkle tree over the roots of the bottom layer trees. This enables parties to agree on all messages of one round making it easier to submit Merkle proofs for messages sent in this round. The message history is composed of the following variables:

$$\begin{aligned}\{\mathsf{mTree}^{\mathsf{round}}_{k^*}\}_{k^*\in[k']} &:= \{\mathsf{MTree}(\{H(\widetilde{\mathsf{msg}}^{(i,j)}_{(\ell,k^*)})\}_{i\in[n],j\neq i})\}_{k^*\in[k']}\\ \mathsf{mTree}_{k'} &:= \mathsf{MTree}(\{\mathsf{MRoot}(\mathsf{mTree}^{\mathsf{round}}_{k^*}\}_{k^*\in[k']})\\ \mathsf{root}^{\mathsf{msg}}_{k'} &:= \mathsf{MRoot}(\mathsf{mTree})\end{aligned}$$

Additionally, if $\mathsf{cert}' \neq \bot$, parties compute the following:

$$\begin{aligned}&\text{(Always):} && \mathsf{initData} := (\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m)}\rangle_m, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m)})\\ &&& \mathsf{root}^{\mathsf{state}} := \langle \mathsf{MRoot}(\mathsf{sTree}_\ell)\rangle_m\\ &\text{(If } k>0\text{):} && \mathsf{state}_{out} := (\mathsf{hash}^{(m)}_{(\ell,k)}, \mathsf{MProof}(\mathsf{hash}^{(m)}_{(\ell,k)}, \mathsf{sTree}_\ell))\\ &\text{(If } k>1\text{):} && \mathsf{state}_{in} := (\mathsf{state}^{(m)}_{(\ell,k')}, \mathsf{MProof}(H(\mathsf{state}^{(m)}_{(\ell,k')}), \mathsf{sTree}_\ell))\\ &&& (\{\mathsf{mTree}^{\mathsf{round}}_{k^*}\}_{k^*\in[k']}, \mathsf{mTree}_{k'}, \mathsf{root}^{\mathsf{msg}}_{k'}) := \mathsf{mesHistory}(k', \ell)\\ &&& \sigma_{k'} := \mathsf{MProof}(\mathsf{MRoot}(\mathsf{mTree}^{\mathsf{round}}_{k'}), \mathsf{mTree}_{k'}))\\ &&& \mathcal{M}_{in} := \{(\widetilde{\mathsf{msg}}^{(j,m)}_{(\ell,k')}, \mathsf{MProof}(H(\widetilde{\mathsf{msg}}^{(j,m)}_{(\ell,k')}), \mathsf{mTree}^{\mathsf{round}}_{k'}))\}_{j\in[n]}\end{aligned}$$

The punishment protocol. The main difficulty of constructing a punishment protocol $\mathsf{Punish}^{\mathsf{sp}}$ for this scenario is that there is no publicly verifiable evidence about messages like a common transcript used in the previous transformations. Hence, incoming messages required for the computation of a particular protocol step cannot be validated directly. Instead, the actions of all parties need to be validated against the emulated actions based on the initial states. This leads to the problem that deviations from the protocol can cause later messages of other honest parties to deviate from the emulated ones as well. Therefore, it is important that the judge disputes the earliest occurrence of misbehavior.

We divide the punishment protocol $\mathsf{Punish}^{\mathsf{sp}}$ into three phases. First, the judge determines the earliest accusation of misbehavior. To this end, if $\mathsf{cert} \neq \bot$ all parties start by sending tuple *ids* from cert to $\mathcal{J}^{\mathsf{sp}}$ and the judge selects the tuple with the smallest indices (ℓ, k, m, i). This mechanism ensures that either the first malicious message or malicious state hash received by an honest party is

[5] Formally, parties need to re-execute the emulation, as we do not allow them to use any data not included in the certificate.

disputed or the adversary blames some party at an earlier point. To look ahead, if the adversary blames an honest party at an earlier point, the punishment will not be successful and the malicious blamer will be punished for submitting an invalid accusation. If the adversary blames another malicious party, either one of them will be punished. This mechanism ensures that if an honest party submits an accusation, a malicious party will be punished, even if it is not the honest party's accusation that is disputed.

If there has not been any accusation submitted in the first phase, $\mathcal{J}^{\mathsf{sp}}$ reimburses all parties. Otherwise, $\mathcal{J}^{\mathsf{sp}}$ defines a blamer P_b, the party that has submitted the earliest accusation, and an accused party P_m. P_b either accuses misbehavior in the initial state, the first round, or in some later round. For the former two, misbehavior can be proven in a straightforward way, similar to our first construction. For the latter, P_b is supposed to submit a proof containing the hash of a tree of the message history up to the disputed round k. P_m can accept or decline the message history depending on whether the tree corresponds to the one emulated by P_m or not. If the tree is accepted, the certificate can be validated as in previous scenarios, with the only difference that incoming messages are validated with respect to the submitted message history tree instead of the common public transcript. In case any party does not respond in time, this party is considered maliciously and is financially punished.

If the message history is declined, the protocol transitions to the third phase. Parties P_b and P_m together with $\mathcal{J}^{\mathsf{sp}}$ execute a bisection search in the message history tree to find the first message they disagree on (cf. Section 5.3). By definition they agree on all messages before the disputed one – we call these messages the *agreed sub-tree*. At this step, $\mathcal{J}^{\mathsf{sp}}$ can validate the disputed message of the history tree (not the one disputed in the beginning) the same way as done in previous constructions with the only difference that incoming messages are validated with respect to the agreed sub-tree.

The number of interactions is logarithmic while the computation complexity of the judge is linear in the protocol complexity. We can further reduce the computation complexity to be logarithmic in the protocol complexity while still having logarithmic interactions using the efficiency improvements described in supplementary materials G. The judge is defined as follows:

Protocol $\mathsf{Punish}^{\mathsf{sp}}$

Phase 1: Determine earliest accusation

1. If $\mathsf{cert} \neq \bot$, P_h sends $ids := (\ell, k, m, i)$ taken from cert to $\mathcal{J}^{\mathsf{sp}}$ which stores (ℓ, k, m, i, h).
2. $\mathcal{J}^{\mathsf{sp}}$ waits till time T to receive message (ℓ, k, m, i) from parties P_b for $b \in [n]$. If no accusations have been received, $\mathcal{J}^{\mathsf{sp}}$ sends d coins to each party at time T. Otherwise, $\mathcal{J}^{\mathsf{sp}}$ picks the *smallest* tuple (ℓ, k, m, i, b) (ordered in this sequence), sets $k' := k - 1$ and continues with Phase 2.

Timeout: If its P_j's turn for $j \in \{b, m\}$ and P_j does not respond with a valid message, i.e., one that is not discarded, in time, P_j is considered malicious and $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_j.

Phase 2: First evidence

3. If $k < 2$, P_b sends $(\mathsf{initData}, \mathsf{root}^{\mathsf{state}}, \mathsf{state}_{out}, \langle \mathsf{msg}^{(m,i)}_{(\ell,k)} \rangle_m)$ taken from cert to $\mathcal{J}^{\mathsf{sp}}$
   (a) $\mathcal{J}^{\mathsf{sp}}$ parses $\mathsf{initData}$ to $(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m)} \rangle_m, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m)})$ and sets $\mathsf{state}_0 = \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(\ell,m)}, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m)})$. If $\mathsf{Verify}(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m)} \rangle_m) = \mathsf{false}$ or $\mathsf{state}_0 = \bot$, $\mathcal{J}^{\mathsf{sp}}$ discards. If $\mathsf{state}_0 = \mathsf{bad}$, $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_m.
   (b) If $\mathsf{Verify}(\langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_m) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
   (c) If $i = 0$ and $\mathsf{wrongState}(\mathsf{state}_0, \bot, \mathsf{state}_{out}, \emptyset, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \bot, \ell, k, m) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
   (d) If $i > 0$, $\mathsf{Verify}(\langle \mathsf{msg}^{(m,i)}_{(\ell,k)} \rangle_m) = \mathsf{false}$ or $\mathsf{wrongMsg}(\mathsf{state}_0, \bot, H(\mathsf{msg}^{(m,i)}_{(\ell,k)}), \emptyset, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \bot, \ell, m, k, i) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
   (e) $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_m.
4. Otherwise, P_b sends $(\mathsf{root}^{\mathsf{state}}, \mathsf{state}_{in}, \mathsf{state}_{out}, \langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_m, \mathsf{root}^{\mathsf{msg}}, \mathsf{root}^{\mathsf{round}}_{k'}, \sigma_{k'}, \mathcal{M}_{in}, \mathsf{msg}_{out})$ taken from cert to $\mathcal{J}^{\mathsf{sp}}$.
   (a) P_m executes $\mathsf{mesHistory}(k-1, \ell)$. Let $\widetilde{\mathsf{root}}^{\mathsf{msg}}$ be the root of the emulated message history tree. If $\mathsf{root}^{\mathsf{msg}} \neq \widetilde{\mathsf{root}}^{\mathsf{msg}}$ P_m sends $\widetilde{\mathsf{root}}^{\mathsf{msg}}$ to $\mathcal{J}^{\mathsf{sp}}$. Otherwise, P_m sends $(\bot)$.
   (b) If $\widetilde{\mathsf{root}}^{\mathsf{msg}}$ received by P_m does not equal $\bot$, $\mathcal{J}^{\mathsf{sp}}$ jumps to phase 3.
   (c) $\mathcal{J}^{\mathsf{sp}}$ checks that $\mathsf{Verify}(\langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_m) = \mathsf{true}$ and $\mathsf{MVerify}(\mathsf{root}^{\mathsf{round}}_{k'}, k', \mathsf{root}^{\mathsf{msg}}, \sigma_{k'}) = \mathsf{true}$ and discards otherwise.
   (d) If $i = 0$ and $\mathsf{wrongState}(\bot, \mathsf{state}_{in}, \mathsf{state}_{out}, \mathcal{M}_{in}, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \mathsf{root}^{\mathsf{round}}_{k'}, \ell, k, m) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
   (e) If $i > 0$, $\mathsf{Verify}(\langle \mathsf{msg}^{(m,i)}_{(\ell,k)} \rangle_m) = \mathsf{false}$ or $\mathsf{wrongMsg}(\mathsf{state}_0, \mathsf{state}_{in}, H(\mathsf{msg}^{(m,i)}_{(\ell,k)}), \mathcal{M}_{in}, , \mathsf{root}^{\mathsf{state}}_{(\ell)}, \mathsf{root}^{\mathsf{round}}_{k'}, \ell, m, k , i) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
   (f) $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_m.

Phase 3: Dispute the message tree

5. Parties P_b, P_m and $\mathcal{J}^{\mathsf{sp}}$ run bisection sub-protocol Π_{BS} on the top-level tree. P_b's input is the tree with root $\mathsf{root}^{\mathsf{msg}}$; P_m's the one with root $\widetilde{\mathsf{root}}^{\mathsf{msg}}$. $\mathcal{J}^{\mathsf{sp}}$ announces public inputs $\mathsf{root}^{\mathsf{msg}}$ and width of $\mathsf{root}^{\mathsf{msg}}$, $width := k'$. The output is the first round they disagree on k_2, the agreed hash $\mathsf{root}^{\mathsf{round}}_{k'_2}$ of leaf with index $k'_2 := k_2 - 1$ and the hash $\mathsf{root}^{\mathsf{round}}_{(b,k_2)}$ of leaf with index k_2 as claimed by P_m.
6. Parties P_m, P_b and $\mathcal{J}^{\mathsf{sp}}$ run bisection sub-protocol Π_{BS} on the low-level tree. Both, P_m and P_b take as input $\mathsf{mTree}^{\mathsf{round}}_{k_2}$ from their certificate. $\mathcal{J}^{\mathsf{sp}}$ announces public inputs $\mathsf{root}^{\mathsf{round}}_{(b,k_2)}$ and the width of the low level tree $width'n \times (n-1)$. The output is the index x of the first message they disagree on and the hash of this message hash_x as claimed by P_m. The index of the sender of the disputed message is $m_2 := \lceil \frac{x}{n-1} \rceil$ and the index of the receiver $i_2 = x \mod (n-1)$ if $m_2 > (x \mod (n-1))$ and $i_2 := (x \mod (n-1)) + 1$ otherwise.

7. Party P_b define variables as follows – variables that are not explicitly defined are set to $\perp$.

$$\begin{aligned}
\text{(Always):}\quad \mathsf{initData}^2 &:= (\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)} \rangle_m, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m_2)}) \\
\mathsf{root}^{\mathsf{state}} &:= \langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_m \\
\text{(If } k_2 > 1\text{):}\quad \mathsf{state}^2_{in} &:= (\mathsf{state}^{(m_2)}_{(\ell,k_2')}, \mathsf{MProof}(H(\mathsf{state}^{(m_2)}_{(\ell,k_2')}), \mathsf{sTree}_\ell)) \\
\mathcal{M}^2_{in} &:= \{(\mathsf{msg}^{(j,m_2)}_{(\ell,k_2')}, \mathsf{MProof}(H(\mathsf{msg}^{(j,m_2)}_{(\ell,k_2')}), \mathsf{mTree}^{\mathsf{round}}_{k_2'}))\}_{j\in[n]}
\end{aligned}$$

and sends $(\mathsf{initData}^2, \langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_m, \mathsf{state}^2_{in}, \mathcal{M}^2_{in})$ to $\mathcal{J}^{\mathsf{sp}}$.

8. $\mathcal{J}^{\mathsf{sp}}$ parses $\mathsf{initData}^2$ to $(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)} \rangle_m, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m_2)})$ and sets $\mathsf{state}^{(m_2)}_{(0)} := \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)}, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m_2)})$. If $\mathsf{Verify}(\langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_m) = \mathsf{false}$, $\mathsf{Verify}(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)} \rangle_m) = \mathsf{false}$ or $\mathsf{state}^{(m_2)}_{(0)} \in \{\perp, \mathsf{bad}\}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
9. If $\mathsf{wrongMsg}(\mathsf{state}^{(m_2)}_{(0)}, \mathsf{state}^2_{in}, \mathsf{hash}_x, \mathcal{M}^2_{in}, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \mathsf{root}^{\mathsf{round}}_{k_2'}, \ell, m_2, k_2, i_2) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
10. $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_m.

7.1 Security

Theorem 2. *Let* $(\pi_3^{\mathsf{pvc}}, \mathsf{Blame}^{\mathsf{pvc}}, \mathsf{Judge}^{\mathsf{pvc}})$ *be an* n*-party publicly verifiable covert protocol computing function* f *with deterrence factor* ϵ *satisfying the view requirements stated in Eq.* (2)*. Further,* π_3^{pvc} *generates a common public transcript of hashes that is only used for* $\mathsf{Blame}^{\mathsf{pvc}}$ *and* $\mathsf{Judge}^{\mathsf{pvc}}$*. Let* π_3 *be a protocol that is equal to* π_3^{pvc} *but does not generate a common transcript and instead of calling* $\mathsf{Blame}^{\mathsf{pvc}}$ *executes the blame procedure explained above (including execution of* $\mathsf{Evidence}$ *and* $\mathsf{Punish}^{\mathsf{sp}}$*). Further, let the signature scheme* $(\mathsf{Generate}, \mathsf{Sign}, \mathsf{Verify})$ *be existentially unforgeable under chosen-message attacks, the Merkle tree satisfies the binding property, the hash function* H *be collision resistant and the bisection protocol* Π_{BS} *be correct. Then, the protocol* π_3*, together with algorithm* $\mathsf{Blame}^{\mathsf{sp}}$*, protocol* $\mathsf{Punish}^{\mathsf{sp}}$ *and judge* $\mathcal{J}^{\mathsf{sp}}$ *satisfies financially backed covert security with deterrence factor* ϵ *according to Definition 1.*

We formally prove Theorem 2 in the supplementary materials I.

8 Evaluation

In order to evaluate the practicability of our protocols, i.e., to show that the judging party can be realized efficiently via a smart contract, we implemented the judge of our third transformation (cf. Section 7) for the Ethereum blockchain and measured the associated execution costs. We focus on the third setting, the verification of protocols with a private transcript, since we expect this scenario to be the most expensive one due to the interactive punishment procedure. Further, we have extended the transformation such that the protocol does not require a public transcript of state hashes.

Our implementation includes the efficiency features described in supplementary materials G. In particular, we model the calculation of each round's and party's computeRound function as an arithmetic circuit and compress disputed calculations and messages using Merkle trees. The latter are divided into 32-byte chunks which constitute the leave of the Merkle tree. The judge only needs to validate either the computation of a single arithmetic gate or the correctness of a single message chunk of a sent or received message together with the corresponding Merkle tree proofs. The proofs are logarithmic in the size of the computation resp. the size of a message. Messages are validated by defining a mapping from each chunk to a gate in the corresponding computeRound function.

In order to avoid redundant deployment costs, we apply a pattern that allows us to deploy the contract code just once and for all and create new independent instances of our FBC protocol without deploying further code. When starting a new protocol instance, parties register the instance at the existing contract which occupies the storage for the variables required by the new instance, e.g., the set of involved parties. Further, we implement the judge to be agnostic to the particular semi-honest protocol executed by the parties – recall that our FBC protocol wraps around a semi-honest protocol that is subject to the cut-and-choose technique. Every instance registered at the judge can involve a different number of parties and define its own semi-honest protocol. This means that the same judge contract can be used for whatever semi-honest protocol our FBC protocol instance is based on, e.g., for both the generation of Beaver triples and garbled circuits. Parties simply define for each involved party and each round the computeRound function as a set of gates, aggregate all gates into a Merkle tree and submit the tree's root upon instance registration.

Table 1: Costs for deployment, instance registration and optimistic execution.

Protocol steps	n	**Cost** Gas	USD
`Deployment`		4 775 k	639.91
`New instance`	2	287 k	38.41
`New instance`	3	308 k	41.30
`New instance`	5	351 k	47.05
`New instance`	10	458 k	61.43
`Honest execution`	2	178 k	23.92
`Honest execution`	3	224 k	30.07
`Honest execution`	5	316 k	42.38
`Honest execution`	10	546 k	73.14

Gates: Number of gates in the circuit of each computeRound function.
Chunks: Number of chunks in each message.
R: Number of communication rounds.
n: Number of parties.

Table 2: Worst-case execution costs.

Gates	**Chunks**	R	n	**Cost** Gas	USD
10	10	10	3	1 780 k	238.58
1 000	10	10	3	2 412 k	323.25
1 M	10	10	3	3 512 k	470.55
1 B	10	10	3	4 782 k	640.75
1 T	10	10	3	6 182 k	828.35
10	10	10	3	1 785 k	239.14
100	100	10	3	2 086 k	279.61
1 000	1 000	10	3	2 422 k	324.55
100	10	10	3	2 081 k	278.91
100	10	10	4	2 223 k	297.86
100	10	10	7	2 442 k	327.29
100	10	10	10	2 659 k	356.34
100	10	10	50	4 764 k	638.35
100	10	3	3	1 878 k	251.65
100	10	10	3	2 074 k	277.88
100	10	100	3	2 403 k	322.04
100	10	1 000	3	2 834 k	379.79

We perform all measurements on a local test environment. We setup the local Ethereum blockchain with *Ganache* (core version 2.13.2) on the latest supported hard fork, Muir Glacier. The contract is compiled to EVM byte code with *solc*

(version 0.8.1, optimized on 20 runs). As common, we measure the efficiency of the smart contracts via its gas consumption – this metric directly translates to execution costs. Further, we estimate USD costs based on the prices (gas to ETH and ETH to USD) on Aug. 20, 2021 [Eth21, Coi21]. For comparison, a simple Ether transfer costs 21,000 gas resp. 2,81 USD.

In Table 1, we display the costs of the deployment, the registration of a new instance and the optimistic execution without any disputes. The costs of these steps only depend on the number of parties. In Table 2, we display the worst-case costs of a protocol execution for different protocol parameters, i.e., complexity of the computeRound functions, message size, communication rounds and number of parties. In order to determine the worst-case costs, we measured different dispute patterns, e.g., disputing sent messages or disputing gates of the computeRound functions, and picked the pattern with the highest costs. The execution costs, both optimistic and worst case, incorporate all protocol steps, incl. the secure funding of the instance. We exclude the derivation of the initial seeds as this step strongly depends on the underlying PVC protocol.

In the optimistic case, the costs of executing our protocol are similar to the ones of [ZDH19]. The authors report a gas consumption of 482 k gas while our protocol consumes between 465 k and 1 M gas, depending on the number of parties – recall that the protocol of [ZDH19] is restricted to the two-party setting. This overhead in our protocol when considering more than two parties is mainly introduced by the fact that [ZDH19] does assume a single deposit while our implementation requires each party to perform a deposit.

Unfortunately, we cannot compare worst-case costs directly, as the protocol of [ZDH19] validates the consistency of a fixed data structure, i.e., a garbled circuit, while our implementation validates the correctness of the whole protocol execution. In particular, [ZDH19] performs a bisection over the garbled circuit while we perform two bisections, first over the message history and then over the computation generating the outgoing messages; such a message might for example be a garbled circuit. Further, [ZDH19] focuses on a boolean circuit, while we model the computeRound function as an arithmetic circuit – as the EVM always stores data in 32-byte words, it does not make sense to model the function as a boolean circuit. Although not directly comparable, we believe the protocol of [ZDH19] to be more efficient for the special case of a two-party garbling protocol, as the protocol can exploit the fact that a dispute is restricted to a single message, i.e., the garbled circuit, and the data structure of this message is fixed such that the dispute resolution can be optimized to said data structure.

Our measurements indicate that the worst-case costs of each scenario are always defined by a dispute pattern that does not dispute a message chunk but a gate of the computeRound functions. This is why the message chunks have no influence on the worst-case execution costs. Of course, this observation might be violated if we set the number of chunks much higher than the number of gates. However, it does not make sense to have more message chunks than gates because each message chunk needs to be mapped to a gate of the computeRound function defining the value of said chunk.

Both, the number of rounds and the number of parties increase the maximal size of the disputed message history and, hence, the depth of the bisected history tree. As the depth of the bisected tree grows logarithmic in the tree size, our protocol is highly scalable in the number of parties and rounds.

Finally, we note that we understand our implementation as a research prototype showing the practicability of our protocol. We are confident that additional engineering effort can further reduce the gas consumption of our contract.

Acknowledgments

The first, third, and fourth authors were supported by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, and by Robert Bosch GmbH, by the Economy of Things Project. The second author was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by ISF grant No. 1316/18.

References

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *IEEE SP*, 2014.

[AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.

[AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, 2012.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, 2014.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, 1990.

[Coi21] CoinMarketCap. Ethereum (ETH) price. https://coinmarketcap.com/currencies/ethereum/, 2021.

[CRR11] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *CCS*, 2011.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In *CRYPTO*, 2020.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[EFS20] Lisa Eckey, Sebastian Faust, and Benjamin Schlosser. Optiswap: Fast optimistic fair exchange. In *ASIA CCS*, 2020.

[Eth21] Etherscan. Ethereum Average Gas Price Chart. https://etherscan.io/chart/gasprice, 2021.

[FHKS21] Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic compiler for publicly verifiable covert multi-party computation. In *EUROCRYPT*, 2021.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.

[HKK+19] Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In *EUROCRYPT*, 2019.

[KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *CCS*, 2014.

[KGC+18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security*, 2018.

[KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In *ASIACRYPT*, 2015.

[SSS21] Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty computation with covert security and public verifiability. *IACR Cryptol. ePrint Arch.*, 2021.

[TR19] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019.

[W+14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS*, 2020.

[ZDH19] Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. In *CCS*, 2019.

Supplementary Materials: Financially Backed Covert Security

A Covert Security

The notion of *covert security with ϵ-deterrent* was introduced by Aumann and Lindell [AL07]. We focus on the *strong explicit cheat formulation* as this is the strongest given formulation. For the sake of completeness, we take the formal definition almost verbatim from [AL07]. As in the standalone model, the notion is defined in the real world/ideal world paradigm. This means, the security of a protocol is shown by comparing the real world execution with an ideal world execution. In the *real world*, the parties jointly compute the desired function f using a protocol π. Let n be the number of parties and let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, where $f = (f_1, \ldots, f_n)$, be the function realized by π. We define for every input vector $\bar{x} = (x_1, \ldots, x_n)$ the output vector $\bar{y} = (f_1(\bar{x}), \ldots, f_1(\bar{x}))$ where party P_i with input x_i obtains the output $f_i(\bar{x})$. During the execution of π, the adversary $\mathcal{A}$ can corrupt a subset $\mathcal{I} \subset [n]$ of all parties. We define $\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa)$ to be the output of the protocol execution π on input $\bar{x} = (x_1, \ldots, x_n)$ and security parameter κ, where $\mathcal{A}$ on auxiliary input z corrupts parties $\mathcal{I}$. We further specify $\mathsf{OUTPUT}_i(\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa))$ to be the output of party P_i for $i \in [n]$.

In contrast, in the *ideal world*, the parties send their inputs to a trusted party $\mathcal{F}$ which computes function f and returns the result. Hence, the computation in the ideal world is correct by definition. The security of π is analyzed by comparing the ideal-world execution with the real-world execution. The ideal world in covert security is slightly adapted in comparison to the standard secure computation model. In covert security, the ideal world allows the adversary to cheat and cheating is detected with some fixed probability ϵ which is called the *deterrence factor*. Let $\epsilon : \mathbb{N} \to [0,1]$ be a function, the execution in the ideal world is defined as follows.

Inputs: Each party obtains an input; the i^{th} party's input is denoted by x_i. We assume that all inputs are of the same length. The adversary receives an auxiliary input z.

Send inputs to trusted party: Any honest party P_j sends its received input x_j to the trusted party. The corrupted parties, controlled by $\mathcal{S}$, may either send their received input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{S}$ and may depend on the values x_i for $i \in \mathcal{I}$ and auxiliary input z. Denote the vector of inputs sent to the trusted party by $\bar{w}$.

Abort options: If a corrupted party sends $w_i = \mathsf{abort}_i$ to the trusted party as its input, then the trusted party sends abort_i to all of the honest parties and halts. If a corrupted party sends $w_i = \mathsf{corrupted}_i$ to the trusted party as its input, then the trusted party sends $\mathsf{corrupted}_i$ to all of the honest parties and halts. If multiple parties send abort_i (resp., $\mathsf{corrupted}_i$), then the trusted party relates

only to one of them (say, the one with the smallest i). If both $\mathsf{corrupted}_i$ and abort_j messages are sent, then the trusted party ignores the $\mathsf{corrupted}_i$ message.

Attempted cheat option: If a corrupted party sends $w_i = \mathsf{cheat}_i$ to the trusted party as its input , then the trusted party works as follows:

1. With probability ϵ, the trusted party sends $\mathsf{corrupted}_i$ to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends $\mathsf{undetected}$ to the adversary along with the honest parties' inputs $\{x_j\}_{j \notin \mathcal{I}}$. Following this, the adversary sends the trusted party output values $\{y_j\}_{j \notin \mathcal{I}}$ of its choice for the honest parties. Then, for every $j \notin \mathcal{I}$, the trusted party sends y_j to P_j.

The ideal execution then ends at this point. If no w_i equals abort_i, $\mathsf{corrupted}_i$ or cheat_i, the ideal execution continues below.

Trusted party answers adversary: The trusted party computes $(y_1, \ldots, y_n) = f(\bar{w})$ and sends y_i to $\mathcal{S}$ for all $i \in I$.

Trusted party answers honest parties: After receiving its outputs, the adversary sends either abort_i for some $i \in \mathcal{I}$, or $\mathsf{continue}$ to the trusted party. If the trusted party receives $\mathsf{continue}$ then it sends y_j to all honest parties P_j ($j \notin \mathcal{I}$). Otherwise, if it receives abort_i for some $i \in \mathcal{I}$, it sends abort_i to all honest parties.

Outputs: An honest party always outputs the message it obtained from the trusted party. The corrupted parties outputs nothing. The adversary $\mathcal{S}$ outputs any arbitrary (probabilistic polynomial-time computable) function of the initial inputs $\{x_i\}_{i \in \mathcal{I}}$, the auxiliary input z, and the messages obtained from the trusted party.

We denote by $\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),\mathcal{I}}(\bar{x}, 1^\kappa)$ the output of the honest parties and the adversary in the execution of the ideal model as defined above, where $\bar{x}$ is the input vector and the adversary $\mathcal{S}$ runs on auxiliary input z.

Definition 2 (Covert security with ϵ-deterrent). *Let f, Π, and ϵ be as above. Protocol Π is said to* securely compute f in the presence of covert adversaries with ϵ-deterrent *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model such that for every $\mathcal{I} \subseteq [n]$, every balanced vector $\bar{x} \in (\{0,1\}^*)^n$, and every auxiliary input $z \in \{0,1\}^*$:*

$$\{\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}} \stackrel{c}{\equiv} \{\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}}$$

We stated the original definition of covert security from [AL07]. For the input-independent settings, where parties have no secret input, the definition needs to be slightly adapted. Furthermore, the original definition requires that the honest parties agree on the aborting party if any corrupted party sends abort_i. This notion of *identifiable abort* can be considered as an independent research area. We refer the reader to [FHKS21] for more details on the adaptations of the covert security definitions.

In order to prevent detecting parties that behave honest except that they might abort as corrupted, the definition of *non-halting detection accurate* was

introduced in [AL07]. The definition uses the notion of a *fail-stop* party which acts semi-honestly, except that it may halt early.

Definition 3. *A protocol π is* non-halting detection accurate *if for every honest party P_j and every honest or fail-stop party P_k the probability that P_j outputs $\mathsf{corrupted}_k$ is negligible.*

B Publicly Verifiable Covert Security

For the sake of completeness, we state the definition of *publicly verifiable covert security* introduced by Asharov and Orlandi [AO12].

While the original notion in [AO12] was presented for two parties, [FHKS21] and [SSS21] extended their definition to the multi-party case. We present the definition almost verbatim from [FHKS21].

In addition to the covert secure protocol π computing some function f, two algorithms Blame and Judge are defined. Blame takes as input the view of an honest party P_i after P_i outputs $\mathsf{corrupted}_j$ in the protocol execution for $j \in \mathcal{I}$ and returns a certificate cert, i.e., $\mathsf{cert} := \mathsf{Blame}(\mathsf{view}_i)$. The Judge-algorithm takes as input a certificate cert and outputs the identity id_j if the certificate is valid that states that party P_j behaved maliciously; otherwise, it returns none to indicate that the certificate was invalid.

Moreover, the protocol π is slightly adapted such that an honest party P_i computes $\mathsf{cert} = \mathsf{Blame}(\mathsf{view}_i)$ and broadcasts it after cheating has been detected. The modified protocol is denoted by π'.

Definition 4 (Covert security with ϵ-deterrent and public verifiability). *Let f, π', Blame, and Judge be as above. The triple $(\pi', \mathsf{Blame}, \mathsf{Judge})$* securely computes *$f$ in the presence of covert adversaries with ϵ-deterrent and public verifiability *if the following conditions hold:*

1. *(**Simulatability**) The protocol π' securely computes f in the presence of covert adversaries with ϵ-deterrent according to the strong explicit cheat formulation (see Definition 2) and non-halting detection accurate (see Definition 3).*
2. *(**Accountability**) For every PPT adversary $\mathcal{A}$ corrupting parties P_i for $i \in \mathcal{I} \subset [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0,1\}^*)^{n+1}$ the following holds:*
   If $\mathsf{OUTPUT}_j(\mathsf{REAL}_{\pi',\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa)) = \mathsf{corrupted}_i$ for $j \in [n] \setminus \mathcal{I}$ and $i \in \mathcal{I}$ then:
   $$\Pr[\mathsf{Judge}(\mathsf{cert}) = \mathsf{id}_i] > 1 - \mu(n),$$
   where cert is the output certificate of the honest party P_j in the execution.
3. *(**Defamation Freeness**) For every PPT adversary $\mathcal{A}$ corrupting parties P_i for $i \in \mathcal{I} \subset [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0,1\}^*)^{n+1}$ and all $j \in [n] \setminus \mathcal{I}$:*
   $$\Pr[\mathsf{Cert}^* \leftarrow \mathcal{A}; \mathsf{Judge}(\mathsf{cert}^*) = \mathsf{id}_j] < \mu(n).$$

C Details about PVC Features

In this section, we provide details about the PVC features described in Section 4.

Cut-and-choose. Although not required per definition of PVC, a fundamental technique used by all existing PVC protocols is the *cut-and-choose* approach that leverages a semi-honest protocol and which comes in two variants. On the one hand, in case of semi-honest protocols where parties do not provide secret inputs (e.g., to set up correlated randomness in preprocessing protocols), all parties execute t instances of the semi-honest protocol. Afterwards, the views (i.e., input and randomness) of all parties are revealed in a subset O of the protocol instances, where $t > |O| \geq s$. This step, which we call *opening* a protocol instance, allows all parties to emulate the execution of the opened instances and compare the computed messages with the messages that have been exchanged during the real execution. Since the behavior of all parties in at least s instances can be checked, the deterrence factor of the resulting PVC protocol is $\epsilon = \frac{s}{t}$. The output of one unopened instance is then used as output of the PVC protocol.

On the other hand, if parties have secret inputs to a semi-honest protocol, input privacy is guaranteed by splitting each input into several shares. More precisely, let n parties, where each party P_i has a secret input x_i jointly compute a semi-honest protocol. Instead of executing t instances of this protocol, just a single instance with $t \cdot n$ parties is executed. Each real party P_i emulates t virtual parties and secret shares its input x_i to these virtual parties. The protocol executed between the virtual parties first reconstructs the private inputs x_i and then computes the desired function f on the reconstructed inputs. In this setting, only a subset O' of each real party's virtual parties is opened. Since the subset O' also satisfies $t > |O'| \geq s$, the resulting deterrence factor in this setting equals $\epsilon = \frac{s}{t}$ as well. Input privacy against passive attacks is guaranteed by ensuring that at least one virtual party per real party is not opened.

While the first setting, where parties do not have private input, is called *input-independent* and is used for preprocessing in many state-of-the-art actively secure protocols [DPSZ12, DKL$^+$13, YWZ20], we call protocols with private inputs *input-dependent*.

Techniques to derive the initial states. The feature of deriving the initial state is explain in Section 4.3. Existing PVC protocol realize this features in one of two ways. On the one hand, parties create signed commitments on their initial states and provide openings after the subset of opened instances are determined. In order to prevent aborting if cheating will be detected, [FHKS21] and [SSS21] utilizes time-lock puzzles that are jointly generated before the subset is revealed to the parties. Even if a party aborts after learning the opened instances, she cannot prevent the other parties from learning the openings on these instances. [FHKS21] adds a verifiability property to the time-lock puzzle that enables parties to generate a proof along with the solution of a puzzle. This proof is utilized to enable efficient verification of the puzzle solution. In case a party puts incorrect opening values into the time-lock puzzles, other parties can show this

inconsistency to a third party. On the other hand, there exists protocols that use oblivious transfer (OT) for this purpose. Introduced by [AO12] and used by [ZDH19, HKK+19, DOS20] in the two-party setting. [DOS20] additionally sketches a multi-party version of a PVC protocol based on OT. The main idea of using a s-out-of-t OT is to reveal the initial state of s instances without revealing the chosen subset to the sender. By letting the sender sign the transcript of the OT, the receiver can reveal his own randomness used during the OT and provide this data as evidence to a third party. The case of inconsistent values as in the first approach cannot happen since the value is directly taken from the output of the OT.

D Building Block Security

In this section, we show security of the building blocks presented in Section 5 and show that incorporating the building blocks into the PVC protocol does not affect the protocol's security.

Internal state commitments. In the described context it is sufficient to create commitments by hashing the internal states. Intuitively, the collision resistance of the hash function ensures that the state hash is binding and the random seeds that are part of the intermediate states ensure that there is sufficient entropy for the hash to be hiding. As the state commitments do not leak information (to computationally bounded adversaries) about the internal states, this modification does neither harm the security of the semi-honest protocol instances nor the security of the PVC protocol. The benefit of this construction is that the commitments do not need to be opened explicitly. Instead, it is sufficient that the validator of the commitments can emulate the protocol execution and compare the resulting intermediate internal states with the received commitments to verify correctness of the commitments. Due to the signatures on the state tree, the validator can prove to a third party that another party has been in a particular state by providing the signature and a Merkle tree proof.

The internal state commitments are not a modification of the PVC protocol but to the underlying semi-honest protocol instances. Hence, it cannot affect the security of the PVC protocol. We show that it does not affect the security of the underlying semi-honest protocol, as well. Concretely, the simulator that is used to prove security of the underlying semi-honest protocol instance would simply forge an arbitrary random state and would provide a hash of this state to the adversary. As the real state incorporates a pseudorandom seed and the forged state is random, the hash of those two are computationally indistinguishable (if the hash function is modelled as a random oracle). This fact allows to proof indistinguishability between two hybrid experiments, one in which the simulator uses forged states and one in which the simulator uses the real state to generate the state hashes. Note that the PVC simulator uses the semi-honest simulator only in the protocol instances that are not opened preventing the PVC simulation to be distinguished from a real world execution based on the opened states.

Signature encoding. The specific signature encoding does not affect security of the PVC protocol because the modification prepends data tuples with public information before being signed – the indices are public and known to every party. Hence, the simulator utilized when proving simulatability of the PVC protocol can apply the encoding itself without having to extract any additional knowledge.

Bisection of trees. We briefly provide an intuition for the correctness of the Bisection protocol, i.e., why the Bisection protocol Π_{BS} indeed outputs the index of the first leaf at which the two trees mTree_b and mTree_m differ. To this end, we make two observations. First, note that the protocol is only executed in case the roots of the trees differ, i.e., $\mathsf{MRoot}(\mathsf{mTree}_b) \neq \mathsf{MRoot}(\mathsf{mTree}_m)$. Second, the values of a node only differ in case the values of at least one child node differ. Based on these observations, in each round, the value of the left child node is request. In case the parties provide different values, the step is repeated for the subtree that has the left child as root. In case the parties provide the same value, this node and the whole subtree including all the leaves must be identical. Furthermore, based on the observation, we know that the values of the right child node must differ. The process is repeated until the lowest layer, i.e., the leaves, are reached. Since always the values of the left child node are checked, this process guarantees that the output is the index of the first leaf at which the two trees differ. It remains to mention that the parties have to provide Merkle proofs under the tree mTree_j for $j \in \{b, m\}$ along with the node value in each round. This ensures that the parties always provide the correct values from their trees and no maliciously chosen values.

E Proof of Theorem 1

Here, we present the security proof of Therorem 1 presented in Section 6.1. The theorem states the security of our transformation from PVC protocols of the first class (i.e., input-independent protocols with public transcript) to FBC protocols.

Proof. Since π^{pp} is a PVC protocol, it already satisfies the simulatability property. In case one starts with a PVC protocol that provides the required features as stated above but has not already the building blocks from Section 5.1 and 5.2 built in, these two building blocks can easily be incorporated (cf. Sections 5.1 and 5.2).

Financial accountability. Note that the punishment protocol $\mathsf{Punish}^{\mathsf{pp}}$ in this construction is non-interactive. Thus, parties submit proofs to $\mathcal{J}^{\mathsf{pp}}$ who punishes a malicious party if the proof is valid and the blamed party has not already been punished. Since the adversary cannot prevent the punishment of a malicious party by trying to punish either an honest party or a malicious party, it follows that we only have to show correctness of the punishment, i.e., that $\mathcal{J}^{\mathsf{pp}}$ interprets the submitted certificate of honest parties correctly except with negligible probability.

Upon detecting cheating and based on the prerequisites stated in Eq. (1a)-(1e), the views of the honest parties either contain inconsistent data for the initial state ($\mathsf{initData}^{\mathsf{core}}$) of some corrupted party or the honest parties have received an invalid message (i.e., one with a wrong state hash, a wrong message hash, or a wrong message).

First, note that all signatures originating from the blamed party provided by the honest party to $\mathcal{J}^{\mathsf{pp}}$ are valid according to the view definitions in Eq. (1a)-(1e). Further, the Merkle Tree proofs are created by the honest party correctly and hence are accepted by $\mathcal{J}^{\mathsf{pp}}$. It follows that $\mathcal{J}^{\mathsf{pp}}$ does not discard any request of honest party due to invalid inputs. In all case where the honest party detects misbehavior during the blame algorithm $\mathsf{Blame}^{\mathsf{pp}}$ (i.e., inconsistent initial state data, wrong state computation, wrong message computations), the smart contract repeats the computation the same way the honest party has executed it locally. Hence, the judge detects cheating the same way the honest party did. This ensures that there is one malicious party P_m losing its security deposit of d coins in case the honest parties detect cheating.

Financial defamation freeness. We first provide an intuition about the proof for financial defamation freeness and then the formal security proof.

Intuitively, the financial defamation freeness property states that the adversary cannot trigger the judge to punish an honest party. Formally, we show financial defamation freeness via a sequence of games. We start with the security game as defined in Section 3 using the punishment protocol $\mathsf{Punish}^{\mathsf{pp}}$ and the judge $\mathcal{J}^{\mathsf{pp}}$. Assume the adversary $\mathcal{A}$ blames honest party P_h in the punishment protocol. $\mathcal{A}$ needs to provide a certificate cert to the judge $\mathcal{J}^{\mathsf{pp}}$ as defined by $\mathsf{Blame}^{\mathsf{pp}}$. Next, we change step-by-step the information used by $\mathcal{J}^{\mathsf{pp}}$ to check for malicious behavior. Concretely, we replace in each game a value provided by the adversary to $\mathcal{J}^{\mathsf{pp}}$ with a value that is given by the challenger. We replace the stated values after being validated but before being used for behavior verification. Each of the values in cert is validated via a cryptographic primitive, i.e., a signature scheme, a Merkle tree or a collision-resistant hash function. This way, we can show that the exchanged values in two adjacent games are identical except with negligible probability via a reduction to the utilized cryptographic primitive. It follows that the success probabilities of adjacent games are negligible close to each other. We end up with a game where the judge $\mathcal{J}^{\mathsf{pp}}$ uses only values provided by the challenger which emulates the honest parties. Since in the final game, $\mathcal{J}^{\mathsf{pp}}$ uses only values from an honest party for verification, it is easy to analyze that the success probability in the final game is zero. Due to the fact that we use a constant number of games, we are able to conclude that the success probability in the original security game is negligible.

For the following formal security proof recall, that $\mathcal{A}$ needs to provide a certificate cert to the judge $\mathcal{J}^{\mathsf{pp}}$ as defined by $\mathsf{Blame}^{\mathsf{pp}}$ that includes the following:

$$
\begin{aligned}
&(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,h)} \rangle_h, \mathsf{initData}^{\mathsf{aux}}_{(\ell,h)}),\\
&\langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_h \text{ and } \langle \mathsf{root}^{\mathsf{msg}}_{(\ell)} \rangle_h,\\
&\mathsf{state}_{in} := (\mathsf{state}_{k-1}, \sigma_{in}),\\
&\mathcal{M}_{in} := \{(\mathsf{msg}^j, \sigma^j)\}_{j \in [n] \setminus \{h\}},\\
&\mathsf{state}_{out} := (\mathsf{hash}_{out}, \sigma_{out}) \text{ and}\\
&\mathsf{msg}_{out} := (\mathsf{hash}, \sigma).
\end{aligned}
$$

Additionally, $\mathcal{J}^{\mathsf{pp}}$ makes use of the initial internal state derived from $\mathsf{initData}$, i.e.,

$$\mathsf{state}_0 = \mathsf{deriveInit}(\mathsf{initData}^{core}_{(\ell,h)}, \mathsf{initData}^{aux}_{(\ell,h)}).$$

Without loss of generality, assume the adversary blames honest party P_h. We replace in a sequence of games value provided by the adversary to $\mathcal{J}^{\mathsf{pp}}$ with values that are given by the challenger. We show that the success probabilities of adjacent games are negligible close to each other via reductions to the underlying primitives, i.e., hash functions, Merkle trees and signatures. We end up with a game where the judge $\mathcal{J}^{\mathsf{pp}}$ uses only values provided by the challenger which emulates the honest parties. We state the games in the following.

- $Game_0$: The original game in which the challenger executes the honest parties and $\mathcal{J}^{\mathsf{pp}}$ as defined by the protocol.
- $Game_1$: In this game, if $\mathsf{Verify}(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,h)} \rangle_h) = \mathsf{true}$ in step 2, we replace $\mathsf{initData}^{\mathsf{core}}_{(\ell,h)}$ provided by the adversary by $\widetilde{\mathsf{initData}}^{\mathsf{core}}_{(\ell,h)}$ as signed by P_h during the protocol execution. As the only correctly encoded initial state data for the disputed instance that is signed by party P_h is $\widetilde{\mathsf{initData}}^{\mathsf{core}}_{(\ell,h)}$, it holds that $\mathsf{initData}^{\mathsf{core}}_{(\ell,h)}$ either equals $\widetilde{\mathsf{initData}}^{\mathsf{core}}_{(\ell,h)}$ or the adversary can forge signatures. Hence, it can be proven that the success probability in $Game_1$ and $Game_0$ is negligible close via a reduction to the unforgeability property of the signature scheme.
- Similar to $Game_1$, we replace further signed data provided by the adversary by the *correct* data taken from the honest parties' views. Negligible close success probability is shown analogous to $Game_1$.
  - $Game_2$: If $\mathsf{Verify}(\langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_h) = \mathsf{true}$ in step 3, we replace $\mathsf{root}^{\mathsf{state}}_{(\ell)}$ provided by the adversary by $\widetilde{\mathsf{root}}^{\mathsf{state}}_{(\ell)}$ as signed by P_h during the protocol execution.
  - $Game_3$: If $\mathsf{Verify}(\langle \mathsf{root}^{\mathsf{msg}}_{(\ell)} \rangle_h) = \mathsf{true}$ in step 3, we replace $\mathsf{root}^{\mathsf{msg}}_{(\ell)}$ provided by the adversary by $\widetilde{\mathsf{msg}}^{\mathsf{state}}_{(\ell)}$ as signed by P_h during the protocol execution.

– $Game_4$: In this game, if $\mathsf{deriveInit}(\mathsf{initData}^{core}_{(\ell,h)}, \mathsf{initData}^{aux}_{(\ell,h)}) \neq \bot$, we replace state_0 calculated as $\mathsf{deriveInit}(\mathsf{initData}^{core}_{(\ell,h)}, \mathsf{initData}^{aux}_{(\ell,h)})$ by $\widetilde{\mathsf{state}}^h_0$ extracted from P_h. By definition (cf. Section 4.3), the adversary cannot find any x such that $\mathsf{deriveInit}(\mathsf{initData}^{core}, x) \notin \{\bot, \mathsf{state}^h_0\}$ expect with negligible probability. Hence, the success probability in $Game_3$ and $Game_4$ is negligible close.
– $Game_5$: In this game, if $\mathsf{MVerify}(\mathsf{hash}, \mathsf{getIndex}(k,h,i), \mathsf{root}^{\mathsf{msg}}_{(\ell)}, \sigma) = \mathsf{true}$ in step 5, we replace hash provided by the adversary by $\widetilde{\mathsf{hash}}$, the hash of the message actually sent by P_h to P_i in round k of instance ℓ. It holds that either $\mathsf{msg} = \widetilde{\mathsf{msg}}$ or the adversary can break the binding property of the Merkle tree. Hence, it can be proven that the success probability in $Game_4$ and $Game_5$ is negligible close via reduction to the security guarantees of the Merkle tree.
– $Game_6$: In this game, if $\mathsf{MVerify}(\mathsf{hash}_{out}, \mathsf{getIndex}(k,h), \mathsf{root}^{\mathsf{state}}, \sigma_{out}) = \mathsf{true}$ in step 1 of function $\mathsf{wrongState}$, we replace hash_{out} provided by the adversary by $\widetilde{\mathsf{hash}}$, the intermediate state hash actually sent by P_h in round k of instance ℓ. Negligible close success probability is shown analogous to $Game_5$
– $Game_7$, $Game_8$ and $Game_9$: We replace further messages provided by the adversary that are verified via a Merkle tree proof by messages taken from the view of the honest parties. In particular, if $\mathsf{MVerify}(H(\mathsf{msg}^j), \mathsf{getIndex}(k,j,h), \sigma^j, \mathsf{root}^{\mathsf{msg}}) = \mathsf{true}$ in step 3b of shared functions $\mathsf{wrongState}$ and $\mathsf{wrongMsg}$, we replace msg^j by $\widetilde{\mathsf{msg}}^j$, the message actually sent by P_j to P_h in round k of instance ℓ. Negligible close success probabiliy is shown in three steps.
  - In $Game_7$, we directly let the adversary win if $H(\mathsf{msg}^j) \neq H(\widetilde{\mathsf{msg}}^j)$. Note that this action is only performed if $\mathsf{MVerify}(H(\mathsf{msg}^j), \mathsf{getIndex}(k,j,h), \sigma^j, \mathsf{root}^{\mathsf{msg}}) = \mathsf{true}$. Hence, it holds that $H(\mathsf{msg}^j) = H(\widetilde{\mathsf{msg}}^j)$ or the adversary can break the binding property of the Merkle tree. Therefore, it can be proven that the success probability in $Game_6$ and $Game_7$ is negligible close via reduction to the security guarantees of the Merkle tree.
  - In $Game_8$, if $H(\mathsf{msg}^j) = H(\widetilde{\mathsf{msg}}^j)$, we replace msg^j by $\widetilde{\mathsf{msg}}^j$. It holds that either $\mathsf{msg}^j = \widetilde{\mathsf{msg}}^j$ or the adversary has found a collision to the hash function. Hence, it can be proven that the success probability in $Game_7$ and $Game_8$ is negligible close via a reduction to the collision resistance of the hash function.
  - In $Game_9$, we remove the modification from $Game_7$. Negligible close success probability is shown analogously.
– $Game_{10}$, $Game_{11}$ and $Game_{12}$: We replace the state for round $k-1$ provided by the adversary that is verified via a Merkle tree proof by the state taken from the view of the honest party. In particular, if $\mathsf{MVerify}(H(\mathsf{state}_{in}), \mathsf{getIndex}(k,j), \sigma_{in}, \mathsf{root}^{\mathsf{state}}) = \mathsf{true}$ in step 3a of shared functions $\mathsf{wrongState}$ and $\mathsf{wrongMsg}$, we replace state^j by $\widetilde{\mathsf{state}}^j$, the state of P_h in round k of instance ℓ. Negligible close success probability is shown analogous to $Game_7$, $Game_8$ and $Game_9$.

Finally, $Game_{12}$ is a security game in which $\mathcal{J}^{\mathsf{pp}}$ does not use any inputs of the adversary for behavior verification. Instead, all inputs are just used to verify validity (e.g., correct signatures or Merkle tree proofs). If any input is not validated correctly, the request is declined and the honest party is not punished. If the inputs are validated correctly, they are all replaced by the correct ones from the view of the honest parties. Based on these inputs, the smart contract checks the behavior of the honest party, i.e., that state_0^h is consistent and that the disputed message or state has been computed by P_h correctly. For the former, the smart contract judge uses $\widetilde{\mathsf{state}}_0^h$ extracted from P_h which cannot be bad. As for the latter, $\mathcal{J}^{\mathsf{pp}}$ takes the values from the view of the blamed and honest party and exactly repeats the honest party's computation, the disputed message respectively state needs to be correct as well. Hence, the judge never punishes an honest party in $Game_{12}$ and hence punishes an honest party in $Game_0$ with negligible probability, which proves financial defamation freeness.

F Class 2: Input-Dependent with Public Transcript

In contrast to the first class, the second class contains input-dependent PVC protocols where all parties possess a common public transcript of messages (cf. Section 4.4). To guarantee input privacy of parties, the cut-and-choose approach for input-dependent protocols is applied (cf. Section 4.1 and supplementary materials C). This means that only a strict subset of each parties' virtual parties is opened. The public transcript of messages means, parties exchange messages instead of message hashes. Input-dependent PVC protocols of this class can be obtained from semi-honest protocols using the input-dependent compilers of Damgård et al.[DOS20] and Faust et al. [FHKS21].

The transformation for protocols of the second class is only slightly different to the transformation of the first class. Intuitively, there are some virtual parties in this class that are not opened, and hence cannot be recomputed during protocol emulation. However, since all parties know about all messages exchanged during the protocol execution, messages originated from unopened virtual parties are just considered correct. This allows the parties to perform the protocol emulation and still detect malicious behavior from opened parties.

In the described PVC protocols, the n' real parties execute a single instance of a $n' \cdot t$-party protocol in which each real party emulates t virtual parties each holding a share of the real party's input (cf. Section 4.1). The encoding of messages and state hashes (cf. Section 5.2) is still applied as before, where the single instance has index 1. Enabling multiple instances based on the techniques of the first class is a trivial extension. To keep the description simple, we interpret all n virtual parties as real parties in the following [6].

In order to apply our construction to a PVC protocol of the second class, π^{ps}, we require for π^{ps} that the actions of each party P_i per protocol instance

[6] It is a straight-forward improvement for practical implementations to reduce the communication overhead by removing redundant data that is sent to several virtual parties belonging to the same real party.

are deterministically determined by a random seed (cf. Section 4.2), all parties receive signed information (e.g., a commitment and decommitment value) to derive initial states of the other parties in some of the protocol instances (cf. Section 4.3), and parties commit to their intermediate internal states during the instance executions in a publicly verifiable way (cf. Section 5.1).

In the follwing, we denote the set of opened parties as O. Formally, whenever some party P_h in round k transitions to a state $\mathsf{state}_{(k)}^{(h)}$ with the outgoing messages $\{\mathsf{msg}_{(k)}^{(h,i)}\}_{i\in[n]\setminus\{h\}}$, then it actually sends the following to P_i:

$$(\mathsf{msg}_{(k)}^{(h,i)}, \{\mathsf{msg}_{(k)}^{(h,j)}\}_{j\in[n]\setminus\{h,i\}}, \mathsf{hash}_{(k)}^{(h)} := H(\mathsf{state}_{(k)}^{(h)})$$

Hence, he view of any honest party P_h includes:

$$\begin{aligned}
&\{(\langle \mathsf{initData}_{(i)}^{core}\rangle_i, \mathsf{initData}_{(i)}^{aux})\}_{i\in O}\\
&\mathsf{sTree} := \mathsf{MTree}(\{\mathsf{hash}_{(k)}^{(i)}\}_{k\in[R],i\in[n]})\\
&\{\langle \mathsf{MRoot}(\mathsf{sTree})\rangle_i\}_{i\in[n]}\\
&\mathsf{trans} := \{\mathsf{msg}_{(k)}^{(i,j)}\}_{k\in[R],i\in[n],j\in[n]\setminus\{i\}}\\
&\mathsf{mTree} := \mathsf{MTree}(\{\mathsf{hash}_{(k)}^{(h,j)} := H(\mathsf{msg}_{(k)}^{(i,j))}\}_{k\in[R],i\in[n],j\in[n]\setminus\{i\}})\\
&\{\langle \mathsf{MRoot}(\mathsf{mTree})\rangle_i\}_{i\in[n]}
\end{aligned} \tag{3}$$

where $(\mathsf{initData}_{(i)}^{core}, \mathsf{initData}_{(i)}^{aux})$ denotes information all parties hold that can be used to derive the initial state of party P_i (cf. Section 4.3).

The blame algorithm. At the end of protocol π^{ps}, all parties execute the blame algorithm $\mathsf{Blame}^{\mathsf{ps}}$ to generate a certificate cert. The resulting certificate is broadcasted and the honest party finishes the execution of π^{ps} by outputting cert. The certificate is generated as follows:

Algorithm $\mathsf{Blame}^{\mathsf{ps}}$

Determine first cheating

1. P_h runs $\mathsf{state}_{(0)}^{(i)} = \mathsf{deriveInit}(\mathsf{initData}_{(i)}^{\mathsf{core}}, \mathsf{initData}_{(i)}^{\mathsf{aux}}), i)$ for each $i \in O$. Let $\mathcal{B}$ be the set of all tuples $(0, m, 0)$ such that $\mathsf{state}_{(0)}^{(m)} = \mathsf{bad}$. If $\mathcal{B} \neq \emptyset$, goto step 4.
2. Otherwise P_h emulates the protocol execution on input the initial states from all opened parties ($P_i : i \in O$) and all messages from the unchecked parties ($P_j : j \in [n] \setminus O$) to obtain the expected messages and the expected intermediate states of all opened parties, i.e., $(\{\widetilde{\mathsf{msg}}_{(k)}^{(i,j)}\}_{k\in[R],i\in O_i,j\in[n]\setminus\{i\}}, \{\mathsf{state}_{(k)}^{(i)}\}_{k\in[R],i\in O_i}) := \mathsf{emulate}_{\pi}^{\mathsf{ps}}(\{\mathsf{state}_{(0)}^{(i)}\}_{i\in O}, \{\mathsf{msg}_{(k)}^{(i,j)}\}_{i\in[n]\setminus O_i,j\in[n]\setminus\{i\}})$.
3. Let $\mathcal{B}$ be the set of all tuples (k, m, i) such that $\widetilde{\mathsf{msg}}_{(k)}^{(m,j)} \neq \mathsf{msg}_{(k)}^{(m,j)}$ or $H(\mathsf{state}_{(k)}^{(m)}) \neq \mathsf{hash}_{(k)}^{(m)}$) – where $\mathsf{msg}_{(k)}^{(m,j)}$ and $\mathsf{hash}_{(k)}^{(m)}$ are extracted from trans or sTree respectively. In case of an invalid state, set $i = 0$.
4. If $\mathcal{B} = \emptyset$ P_h outputs $\mathsf{cert} := \perp$. Otherwise, P_h picks the tuple (k, m, i) from $\mathcal{B}$ with the smallest k, m, i in this sequence, sets $k' := k - 1$ and defines variables

as follows – variables that are not explicitly defined are set to $\perp$.

$$
\begin{aligned}
\text{(Always):} \quad ids &:= (1, k, m, i) \\
\mathsf{initData} &:= (\langle \mathsf{initData}^{core}_{(m)} \rangle_m, \mathsf{initData}^{aux}_{(m)}) \\
\mathsf{root}^{\mathsf{state}} &:= \langle \mathsf{MRoot}(\mathsf{sTree}) \rangle_m \\
\mathsf{root}^{\mathsf{msg}} &:= \langle \mathsf{MRoot}(\mathsf{mTree}) \rangle_m \\
\text{(If } k > 0\text{):} \quad \mathsf{state}_{out} &:= (\mathsf{hash}^{(m)}_{(k)}, \mathsf{MProof}(\mathsf{hash}^{(m)}_{(k)}, \mathsf{sTree})) \\
\mathsf{msg}_{out} &:= (\mathsf{hash}^{(m,i)}_{(k)}, \mathsf{MProof}(\mathsf{hash}^{(m,i)}_{(k)}, \mathsf{mTree})) \\
\text{(If } k > 1\text{):} \quad \mathsf{state}_{in} &:= (\mathsf{state}^{(m)}_{(k')}, \mathsf{MProof}(H(\mathsf{state}^{(m)}_{(k')}), \mathsf{sTree})) \\
\mathcal{M}_{in} &:= \{(\mathsf{msg}^{(j,m)}_{(k')}, \mathsf{MProof}(H(\mathsf{msg}^{(j,m)}_{(k')}), \mathsf{mTree}))\}_{j \in [n]}
\end{aligned}
$$

5. Output $\mathsf{cert} := (ids, \mathsf{initData}, \mathsf{root}^{\mathsf{state}}, \mathsf{root}^{\mathsf{msg}}, \mathsf{state}_{in}, \mathcal{M}_{in}, \mathsf{state}_{out}, \mathsf{msg}_{out})$.

The punishment protocol. The punishment algorithm $\mathsf{Punish}^{\mathsf{ps}}$ and the judge $\mathcal{J}^{\mathsf{ps}}$ are equal to the one of class 1. Variable t describing the number of instances equals 1.

Security. As $\mathsf{Punish}^{\mathsf{ps}}$ is the same protocol than in class 1, Theorem 3 can be proven to be correct the same way as Theorem 1.

Theorem 3. *Let $(\pi^{\mathsf{ps}}, \cdot, \cdot)$ be an n-party publicly verifiable covert protocol computing function f with deterrence factor ϵ satisfying the view requirements stated in Eq. (3). Further, let the signature scheme $(\mathsf{Generate}, \mathsf{Sign}, \mathsf{Verify})$ be existentially unforgeable under chosen-message attacks, the Merkle tree satisfies the binding property and the hash function H be collision resistant. Then the protocol π^{ps} together with algorithm $\mathsf{Blame}^{\mathsf{ps}}$, protocol $\mathsf{Punish}^{\mathsf{ps}}$ and judge $\mathcal{J}^{\mathsf{ps}}$ satisfies financially backed covert security with deterrence factor ϵ according to Definition 1.*

G Single Gate Validation

We presented three constructions of FBC protocols based on PVC protocols with different features. While we incorporated the major techniques required for these transformations in our protocol specifications, we excluded important efficiency improvements for the sake of presentation.

In the worst case, the specified protocols require the judge to receive an internal state of a party, a set of incoming messages and to compute a full protocol step, i.e. the $\mathsf{computeRound}$ function. Depending on the complexity of the computed step, the amount of data received and the amount of computation executed by the judge can still be massive. To this end, we present an improvement that enables the judge to eventually (i) compute just a single instruction of the $\mathsf{computeRound}$ function (e.g., a simple addition) and (ii) receive only the data required for this particular instruction. Although the technique is applicable to

any kind of program description, we describe it based on an arithmetic circuit. We make use of the bi-section technique (cf. Section 5.3).

First, parties divide both states and messages into individual data units – in our case integers. Instead of hashing states or messages, parties create Merkle trees over the state resp. message chunks and use the roots of these trees as they used the hashes of states resp. messages before. In the following we will refer with the term *message* to both state and messages. We call the trees generated over the message chunks *message trees*.

Second, we interpret the program computing the computeRound function as an arithmetic circuit. In order to define these circuits, parties derive a program description from the definition of the executed semi-honest protocol, which defines for each round and each party the gates of the corresponding computeRound function. These gates state for each wire of the circuit how the wire is computed, e.g., wire z is the addition of wire x and wire y. Further, for each input and output wire, the program description specifies two indices i and j stating that the particular wire needs to equal the i-th message chunk of the message received by P_j respectively the message that is to be send to P_j. The message that a party sends to itself is its intermediate state. Prior to executing the FBC protocol, parties aggregate the protocol description into a Merkle tree and register the root of this tree at the smart contract. We call the generated tree *program tree*.

Once there is a dispute about the result of a computeRound function, parties aggregate the values of all wires of their computation of the computeRound function into another Merkle tree, the *computation tree*. The wires are ordered such that for each gate, the gate's inputs have smaller indices than the gate's output. Next, the disputing parties execute a bi-section search over the computation tree to determine the first wire they disagree on while directly agreeing on all previous wires. Next, the blamer can send the gate corresponding to the disputed wire as well as the inputs to the gate to the smart contract. The gate is validated via a Merkle tree proof in the program tree. Depending on whether the wire is an input wire or an intermediate wire, it is validated differently. In case of input wires, the gate specifies the message chunk which should be written on the disputed wire. Note that when disputing a computeRound function, parties have already agreed on all prior messages. This means that the blamer can first submit a message root together with a Merkle proof showing that the message is at the correct position of the agreed message history and then submit a message chunk together with a Merkle proof showing that the chunk is at the position specified by the input-gate. In case of intermediate wires, the blamer submits the values of the ingoing wires together with Merkle proofs showing that these values are at the correct position in the list of agreed wire values. In case there is no dispute in the calculation tree, the blamer can prove that one of the outgoing wires does not correspond to a sent message, the same way parties prove correctness of input wires.

Finally, we want to note that the same technique can be applied to the calculation of the deriveInit function.

H Security Games for Financially Backed Covert Security

In this section, we present graphical representations of the security games for financial accountability $\mathsf{Exp}^{\mathsf{FA}}$ and financial defamation freeness $\mathsf{Exp}^{\mathsf{FDF}}$. The games are described in detail in Section 3.3. Figure 1 contains an illustration of the security game $\mathsf{Exp}^{\mathsf{FA}}$ and Figure 2 illustrates the security game $\mathsf{Exp}^{\mathsf{FDF}}$.

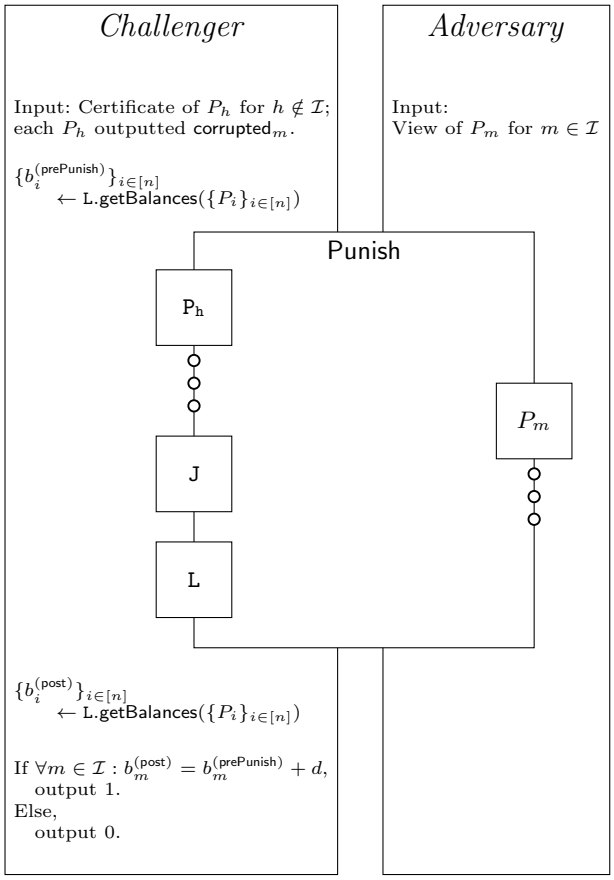

Fig. 1: Security game for financial accountability

I Proof of Theorem 2

Here, we present the security proof of Therorem 2 presented in Section 7.1.

Proof. In order to prove security, we need to show simulatability, financial accountability and financial defamation freeness.

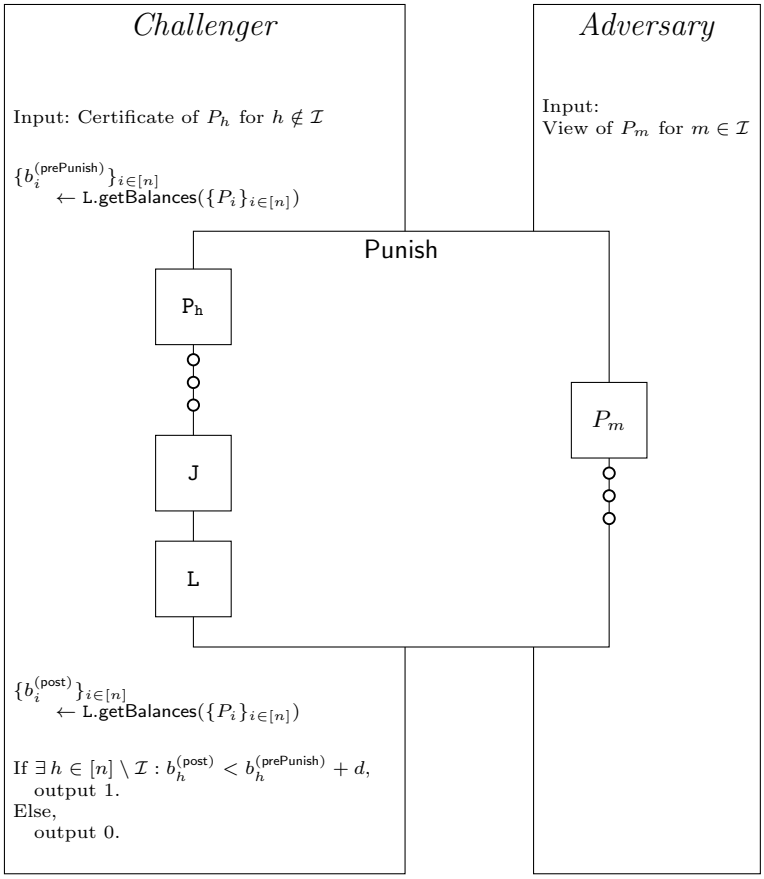

Fig. 2: Security game for financial defamation freeness

Simulatability. Again, simulatability follows from the simulatability of the PVC protocol. Removing the public transcript does not affect simulatability as the transcript is only used by $\mathsf{Blame}^{\mathsf{pvc}}$ and $\mathsf{Judge}^{\mathsf{pvc}}$. However, the extension of the blame procedure requires an extension of the simulation proof. Since the covert functionality ensures that an honest party never outputs $\mathsf{corrupted}_i$ for i denoting another honest party, i.e., $i \in [n] \setminus \mathcal{I}$, π_3 must ensure the same. In our extension, a partial certificate forged by a malicious party could cause an honest party to interpret another honest party to be corrupted. Intuitively, this is not possible because the signed message included in the partial certificate is either the one actually sent by the honest party that cannot be interpreted to be the first wrong message or the adversary has forged a signature. Formally, the ideal world simulator would throw an event bad if the adversary sends a partial certificate cert' that contains a valid signature on a message that has not been sent by the accused honest party. Hence, only partial certificates that can be computed based on the views of the honest parties and partial certificates that are generated by the adversary accusing a corrupted party are considered to derive the index m of the corrupted party. In an hybrid experiment along the transition to the real world execution, the simulator would derive the index m of the corrupted party as in the real world, hence, considering all partial certificate of the adversary containing valid signatures. Computational indistinguishability between the two experiments is guaranteed if the event bad happens only with negligible probability. This can be shown via a reduction to the security guarantees of the signature scheme.

Financial accountability. We split the security proofs conditioned on the occurrence of certain events, e.g., P_m accepting the message history submitted by P_b in step 4. For each resulting case, we individually prove that the adversary has a negligible win chance in the financial accountability game defined in Section 3. Further, we consider only adversaries that always send valid messages, those that are not discarded, to the judge. As discarded messages do not have an influence on the execution, every adversary can be translated into one that sends only valid messages.

In phase one of the punishment protocol, the judge $\mathcal{J}^{\mathsf{sp}}$ determines the earliest accusation, i.e., the first round k of a particular instance in which any party P_b claims that another party P_m has misbehaved. Note that if any party submits an accusation, the judge will eventually punish P_b or P_m. This is due to the fact that $\mathsf{Punish}^{\mathsf{sp}}$ terminates with a finite number of steps and punishes either P_b or P_m when terminating. Each step can only last a fixed amount of time – if a party does not respond in time the judge terminates with punishing that party.

We first split the analysis conditioned on P_b and P_m being malicious or honest. It follows from simulatability that no honest party blames another honest party, i.e., that P_b and P_m cannot both be honest. If both P_m and P_b are malicious, either of them will be punished as explained above. It follows from financial defamation freeness as shown below that if P_b is malicious and P_m is honest then P_m will not be punished except with negligible probability. Hence,

P_b will be punished except with negligible probability. It remains to analyze the case where P_b is honest and P_m is malicious.

We continue with considering the two events that $k < 2$ in step 3 and that $k \geq 2$ but P_m accepts the message history, i.e. $\widetilde{\mathsf{root}}^{\mathsf{msg}} = \bot$ in step 4b. For both cases, we can show similar to the proof in Scenario 1, that P_m is always punished. This is due to the fact, that the inputs of the honest P_b are always accepted (validated successfully) and the judge repeats the validation executed by the honest party to detect the disputed misbehavior, hence, detecting misbehavior of P_m itself.

Next, we continue with the event that $k \geq 2$ but P_m declines the message history. The first bisection protocol determines the disputed round k_2 and ensures that the message tree of round $k_2' := k_2 - 1$ (if any) is the one emulated by P_b and used by P_b for behavior verification. The second bisection protocol determines the concrete message that is disputed. Since both parties agreed on the messages up until this message and the honest P_b correctly compute the next message, the message of P_m must be incorrect. Again, we can show that inputs of the honest P_b are always accepted (validated correctly) and that the judge repeats the validation executed by the honest party to detect the disputed misbehavior, hence, detecting misbehavior of P_m itself.

As the analyzed cases cover all possible cases and the adversary cannot win the financial accountability game in any of the cases with probability higher than negligible, the adversary wins the financial accountability game with at most negligible probability.

Financial defamation freeness. Symmetrically to the financial accountability proof, we split the security proof conditioned on the occurrence of certain events and consider only adversaries that always sent valid messages, i.e., messages that are not discarded.

Again, we split the game according to the event of P_b and P_m being honest or malicious. It follows from simulatability that no honest party blames another honest party, i.e., that P_b and P_m cannot both be honest. If both P_b and P_m are malicious, the judge does not punish any honest party. As shown in the financial accountability game, if P_b is honest and P_m is malicious, P_m is punished with probability negligible close to one which ensures that P_b is not punished.

It remains to show the case of P_b being malicious and P_m being honest. Again, we split this case in further sub-cases. First, we consider the two events that $k < 2$ in step 3 and that $k \geq 2$ but P_m accepts the message history, i.e. $\widetilde{\mathsf{root}}^{\mathsf{msg}} = \bot$ in step 4b. In this case, the punish protocol is analog to the one of Scenario 1: The blamer submits data that is validated via signatures and Merkle tree proofs and used to check the behavior of the blamed honest party. Hence, security can be proven similar to Scenario 1 (cf. Section 6.1) via a sequence of games gradually replacing the data used for behavior verification submitted by the adversary by the data used by the honest party in the real protocol instance to determine her own behavior. Negligible close success probabilities in adjacent games is shown via reductions to the underlying primitives. For instance, if $\mathsf{Verify}(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m)} \rangle_m) = \mathsf{true}$, then $\mathsf{initData}^{\mathsf{core}}_{(\ell,m)}$ provided by the adversary is

either the same value as the one known and signed by P_m or the adversary can forge signatures. In the final game, the judge repeats exactly the computation executed by P_m in the real protocol instance to determine her own behavior. Hence, it is easy to see in the final game that P_m is not punished. It follows that in the original game, P_m is not punished expect with negligible probability. In case $k \geq 2$ but P_m accepts the message history, the messages provided by P_b are exactly the ones received by P_m during the protocol execution. Otherwise, P_m would have received in incorrect message and would have submitted an earlier accusation.

We continue with the event that $k \geq 2$ but P_m declines the message history. It follows from the correctness of the bisection protocol that the message tree of round $k_2' := k_2 - 1$ (if any) correspond to the one emulated by P_m. Further, it follows that the disputed message hash hash_x received by the judge as a result of the bisection protocol, is the one emulated by P_m. The next step, is again symmetric to the one of Scenario 1: The blamer submits data that is validated via signatures and Merkle tree proofs and used to check the behavior of the disputed party P_{m_2}. Hence, we can again prove security via a sequence of games, gradually replacing the data used to calculate the next message of P_{m_2} by the one used by the honest P_m to emulate the next message of P_{m_2}. In the final game, it is obvious that the judge does not punish P_m because the judge uses the same data as P_m to emulate the disputed message of P_{m_2} which ensures that the emulated message hash equals hash_x. It follows that in the original game, P_m is not punished expect with negligible probability.

As the analyzed cases cover all possible cases and the adversary cannot win the financial defamation freeness game in neither of the cases with probability higher than negligible, the adversary wins the financial defamation freeness game with at most negligible probability.

J Shared Judge Logic

In this section, we present methods used across all of our judge specifications. We extracted the common methods from the main body to keep the protocol description compact. For the sake of completeness, we give the formal specifications of these methods in the following:

Judge Shared Logic

$\mathsf{wrongState}(\mathsf{state}_0, \mathsf{state}_{in}, \mathsf{state}_{out}\mathcal{M}_{in}, \mathsf{root}^{\mathsf{state}}, \mathsf{root}^{\mathsf{msg}}, \ell, k, m)$:

1. Parse state_{out} to $(\mathsf{hash}_{out}, \sigma_{out})$. If $\mathsf{MVerify}(\mathsf{hash}_{out}, \mathsf{getIndex}(k, m), \sigma_{out}, \mathsf{root}^{\mathsf{state}}) = \mathsf{false}$, return false.
2. If $k = 1$, set $\mathsf{state}_{(k-1)} = \mathsf{state}_0$ and $\mathcal{M}_{(k-1)} := \{\bot\}_{j \in [n] \setminus \{m\}}$.
3. If $k > 1$, do:
   (a) Parse state_{in} to $(\mathsf{state}_{k-1}, \sigma_{in})$. If $\mathsf{MVerify}(H(\mathsf{state}_{k-1}), \mathsf{getIndex}(k-1, m), \sigma_{in}, \mathsf{root}^{\mathsf{state}}) = \mathsf{false}$, return false.
   (b) Parse $\mathcal{M}_{in}$ to $\{\mathsf{msg}^j, \sigma^j\}_{j \in [n] \setminus \{m\}}$. If $\mathsf{MVerify}(H(\mathsf{msg}^j), \mathsf{getIndex}(k, j, m), \sigma^j, \mathsf{root}^{\mathsf{msg}}) = \mathsf{false}$ for any $j \in [n] \setminus \{m\}$, return false. Otherwise, set $\mathcal{M}_{(k-1)} := \{\mathsf{msg}^j\}_{j \in [n] \setminus \{m\}}$.

4. Calculate $(\mathsf{state}', \cdot) := \mathsf{computeRound}_k^m(\mathsf{state}_{k-1}, \mathcal{M}_{(k-1)})$. If $H(\mathsf{state}') = \mathsf{hash}_{out}$, return false.
5. Return true

$\mathsf{wrongMsg}(\mathsf{state}_0, \mathsf{state}_{in}, \mathsf{hash}_{out}, \mathcal{M}_{in}, \mathsf{root}^{\mathsf{state}}, \mathsf{root}^{\mathsf{msg}}, \ell, k, m, i)$:

1. If $k = 1$, set $\mathsf{state}_{(k-1)} = \mathsf{state}_0$ and $\mathcal{M}_{(k-1)} := \{\bot\}_{j \in [n] \setminus \{m\}}$.
2. If $k > 1$, do:
   (a) Parse state_{in} to $(\mathsf{state}_{k-1}, \sigma_{in})$. If $\mathsf{MVerify}(H(\mathsf{state}_{k-1}), \mathsf{getIndex}(k-1, m), \sigma_{in}, \mathsf{root}^{\mathsf{state}}) = \mathsf{false}$, return false.
   (b) Parse $\mathcal{M}_{in}$ to $\{\mathsf{msg}^j, \sigma^j\}_{j \in [n] \setminus \{m\}}$. If $\mathsf{MVerify}(H(\mathsf{msg}^j), \mathsf{getIndex}(k, j, m), \sigma^j, \mathsf{root}^{\mathsf{msg}}) = \mathsf{false}$ for any $j \in [n] \setminus \{m\}$, return false. Otherwise, set $\mathcal{M}_{(k-1)} := \{\mathsf{msg}^j\}$.
3. Calculate $(\cdot, \{\mathsf{msg}^{(m,j)}\}_{j \in [n] \setminus \{m\}}) := \mathsf{computeRound}_k^m(\mathsf{state}_{k-1}, \mathcal{M}_{(k-1)})$. If $H(\mathsf{msg}^{(m,i)}) = \mathsf{hash}_{out}$, return false.
4. Return true

$\mathsf{getIndex}(k, i, j)$:

1. Set $x := (i-1) \cdot (n-1) + j$.
2. In Scenario 1 and Scenario 2 ($\mathcal{J}^{\mathsf{pp}}$ / $\mathcal{J}^{\mathsf{ps}}$), set $x := x + (k-1) \cdot n \cdot (n-1)$
3. If $(j > i)$, $x := x - 1$.
4. Return x.

$\mathsf{getIndex}(k, i)$: Return $(k-1) \cdot n + i$