The Brave New World of Global Generic Groups and UC-Secure Zero-Overhead SNARKs

Jan Bobolz[1], Pooya Farshim[2,3], Markulf Kohlweiss[1,4], Akira Takahashi[5]

[1] University of Edinburgh, UK
[2] IOG, Switzerland
[3] Durham University, UK
[4] IOG, UK
[5] JPMorgan AI Research & AlgoCRYPT CoE, US

Abstract. The universal composability (UC) model provides strong security guarantees for protocols used in arbitrary contexts. While these guarantees are highly desirable, in practice, schemes with a standalone proof of security, such as the Groth16 proof system, are preferred. This is because UC security typically comes with undesirable *overhead*, sometimes making UC-secure schemes significantly less efficient than their standalone counterparts.

We establish the UC security of Groth16 without any significant overhead. In the spirit of global random oracles, we design a *global (restricted) observable generic group* functionality that models a natural notion of observability: computations that trace back to group elements derived from generators of other sessions are observable. This notion turns out to be surprisingly subtle to formalize. We provide a general framework for proving protocols secure in the presence of global generic groups, which we then apply to Groth16.

jan.bobolz@ed.ac.uk
pooya.farshim@gmail.com
markulf.kohlweiss@ed.ac.uk
takahashi.akira.58s@gmail.com

Table of Contents

1 Introduction

Composable treatments of cryptosystems measure the security of the system under arbitrary attacks relative to those on an ideal version of the system. Various composition theorems then show that if these systems are sufficiently close, the ideal system can be safely replaced by the real system in a wide variety of contexts. A notable framework that formalizes this approach is that of Universal Composability (UC) [Can01, Can20a], which has been widely used in the literature, although other approaches also exist [Mau10, Küs06, HS15, BDF$^+$18].

Unfortunately, to date such composable treatments of security due to their complexity often result in complex and less efficient protocols. This is somewhat dissatisfying as it is exactly simple and efficient cryptosystems (proven in stand-alone models of security) that are widely deployed and thus in need of composable security guarantees. This state of affairs necessitates composable treatment of practical cryptosystems with minimal, preferably no, overhead.

A notable example is that of succinct non-interactive arguments of knowledge (SNARKs), which exactly fall into this gap between composition and usage in complex environments: practical SNARKs are either analyzed under property-based definitions, or else need to be modified or compiled, which increase overheads both in terms of proof sizes and prover/verifier time. Such overheads potentially prevent adoption in practice.

Practical SNARKs are typically proven secure in idealized models of computation, such as the random-oracle model or the generic-group model.[6]

While simple and elegant, these proof techniques do not necessarily lend themselves to composable treatments. The central conflict in that scenario is that both systems' security proofs require exclusive access to the same idealized resource. For instance, it may be that the extractors for two SNARK systems may want to program $\mathcal{H}(0)$ to different conflicting values, or that the knowledge extractor for one system needs to observe all random oracle queries, while that for another needs to keep its oracle queries/programming secret for an indistinguishable simulation [CF24, LR22b]. In such scenarios, we cannot say anything meaningful about the security of either SNARK when composed with the other.

The examples above demonstrate the need for appropriate formalization of idealized models that are compatible with composability. Here we adopt Canetti's UC framework [Can01, Can20a]. For random oracles, this question has been largely solved in the form of the *restricted observable global random oracle* functionality [CJS14, CDG$^+$18], as well as its programmable version $\mathcal{G}$-$\texttt{rpoRO}$ [CDG$^+$18]. The functionality $\mathcal{G}$-$\texttt{roRO}$ works like a globally accessible random oracle and is not exclusively controlled by any UC simulator (as would be the case for a local version). Instead, $\mathcal{G}$-$\texttt{roRO}$ implements an interface through which UC simulators can *partially* control the functionality in the form of observability: the simulator for the protocol running in session sid is able to observe all random oracle queries $\mathcal{H}(sid, \cdot)$ prefixed with sid that are made in protocol sessions $sid' \neq sid$. ($\mathcal{G}$-$\texttt{rpoRO}$ comes with an analogous programming interface.) Details are discussed in Appendix C.

With this mechanism, the single random-oracle resource can be shared among multiple protocols in a way that still gives appropriate control over the resource to the UC simulator (observations, or, in the case of $\mathcal{G}$-$\texttt{rpoRO}$, programming) to enable UC simulation, and in turn composition with other protocols.

In this case, every protocol session sid gets its own hash prefix sid, and while every protocol session is using the same resource $\mathcal{G}$-$\texttt{roRO}$, they can do so with sufficient domain separation so as to not interfere with each other. As a result, we can prove many proof systems and SNARK constructions UC-secure in the presence of $\mathcal{G}$-$\texttt{roRO}$ [LR22b, LR22a, GKO$^+$23, CF24].

[6] Alternatively, they are proven using knowledge assumptions or in the algebraic-group model (AGM).

In contrast to RO-based proof systems, the state of affairs for systems whose proofs of knowledge extraction rely on idealized groups is much less clear. For example, the popular Groth16 SNARK [Gro16], has a security proof that is "close" to UC in the sense it has the prerequisite simulation-extractability and straight-line extraction properties. Despite this, its extraction strategy is not easily compatible with composable frameworks such as the UC framework.

This has led to the strategy of applying some transformation or compilation to SNARKs in order to render their extraction strategy UC-compatible. The cost, however, is increased overhead: one has to accept a noticeable loss in computational efficiency [GKO+23], or sometimes even forgo succinctness [KZM+15]; see Appendix A for an overview of the state of the art.

Our goal is to avoid such overheads and prove practical SNARKs such as Groth16 secure in a composable framework as-is, while using their native extraction strategies. For Groth16-style SNARKs specifically, we have standalone (non-composable) analyses in the GGM [Gro16], in the AGM [FKL18, BKSV21], and under knowledge assumptions [GM17, BFHK23]. However, it is unclear how these analyses apply to a composable setting. Even worse, in contrast to the random-oracle model, it is not even a settled question how to model composable versions of group-related idealized resources. One may consider the following existing approaches:

1. Prove Groth16 secure in the $\mathcal{F}$-$\texttt{GG}$-hybrid model, where $\mathcal{F}$-$\texttt{GG}$ (e.g., [CNPR22]) is simply an ideal functionality implementing a (local) generic group.
2. Prove Groth16 secure in the UC-AGM [ABK+21] framework, which is a composable version of the algebraic-group model. UC-AGM is implemented as a modification of the UC framework whereby adversaries are required to output the discrete-logarithm representations of the group elements that they output in terms of input elements they have received so far.
3. Prove Groth16 secure through [KKK21], which is a composable version of knowledge assumptions. It is implemented as a variant of the Constructive Cryptography [Mau10] framework, where all parties are forced to register the discrete-logarithm representations in terms of their input elements with a global registry whenever they output a group element.

The first option is certainly feasible and a Groth16 proof in the $\mathcal{F}$-$\texttt{GG}$-hybrid model would be considered a folklore adaptation of the standalone Groth16 generic-group security proof [Gro16]. However, the interpretation of $\mathcal{F}$-$\texttt{GG}$ in practice is that every instance of Groth16 (and any other protocol) needs its own independent (generic) group. Of course, this is far from practice, where a few standard groups (such as BLS12-381) are shared among all sessions for many protocols. It is also not desirable from a design standpoint, as the building blocks of complex protocols usually share the same group for compatibility reasons.

The second option, using the UC-AGM [ABK+21], is more reasonable: multiple UC-AGM protocols can share the same group. One of the central conflicts that arise when composing multiple protocols over the same group occurs when group elements output by one protocol or session are used as input to another protocol or session. The outputting protocol is interested in hiding the element's discrete-logarithm representation from the environment (e.g., as part of a simulation strategy), while the receiving protocol is interested in learning the element's discrete-logarithm representation (e.g., for proof of knowledge extraction).

This conflict manifests in two different ways in the UC-AGM. First, the *environment* in the UC-AGM is *not* required to output a discrete-logarithm representation when it provides input to honest parties, say an honest Groth16 verifier. For our interests, this means that the environment can submit a Groth16 proof to the ideal functionality $\mathcal{F}$-$\texttt{NIZK}$ for verification without having to provide a representation. The lack of a representation makes it impossible for the UC simulator to extract a witness, even if the proof was computed honestly by the environment. As a consequence, the UC-AGM is too lenient on the environment, making it unsuitable for *non-interactive* proof systems. In particular, it is unclear how to use it to prove Groth16 UC-secure. Second, the *adversary* in the UC-AGM *is* required to output representations whenever it provides input to any functionality (e.g., sending a network message). This leads to situations where the framework is *too strict*: the adversary may want to use a group element output by one protocol to attack another protocol, but because the adversary (usually) does not know an appropriate discrete-logarithm representation, it is prohibited from using the group element. This means that the framework effectively forbids adversaries from mounting cross-session attacks, meaning taking a group element from one session/protocol to mount an attack against another session/protocol. As a consequence, the UC-AGM is not able to adequately model arbitrary environment/attacker behavior, which is a major downside. We discuss an example in Appendix B. The original UC-AGM paper discusses and explores the shortcomings of the AGM when it comes to composability [ABK+21, Section 1.1], noting that cross-session attacks seem to be an inherent limitation of the AGM rather than a modeling artifact of the UC-AGM.

The third option [KKK21] is similar to the UC-AGM in spirit in that it models algebraic behavior. While [KKK21] is highly configurable and supports a range of different settings, the authors identify inherent conflicts when it comes to composing multiple knowledge assumptions, which roughly correspond to cross-session attacks mentioned above. They conclude that group reuse between multiple protocols remains an open challenge.

This leaves open the question of formulating an adequate framework for composable treatment of security which (1) permits modeling multiple protocols using the same group, (2) does not unnaturally restrict the environment's/adversary's ability to take elements output by one protocol, optionally operate on them, and use the result to attack another protocol, and (3) is suitable to prove modern SNARKs in idealized models for groups, such as Groth16, secure.

1.1 Our contributions

Driven by the fact that the UC-AGM framework (as well as the work of [KKK21] which follows a similar approach to the AGM) have inherent composability shortcomings, we turn our attention to the generic-group model. We propose a new ideal functionality $\mathcal{G}\text{-}\mathtt{oGG}$ (Section 3), in the standard UC framework without modifications, that formalizes access to a *restricted observable global generic (bilinear) group* resource. Similar to its random-oracle counterpart [CJS14, CDG+18], $\mathcal{G}\text{-}\mathtt{oGG}$ works like a globally accessible generic group, but additionally offers an observability interface, which allows simulators, based on domain separation, to observe certain group operations. $\mathcal{G}\text{-}\mathtt{oGG}$ allows for group reuse among multiple protocols, and it does not restrict the environment from using group elements output by one protocol as input to another. Additionally, $\mathcal{G}\text{-}\mathtt{oGG}$ naturally features oblivious sampling of group elements with unknown discrete logarithms. As observed in the literature [LPS23, BFHK23], this is an important feature of real-world groups, realized, e.g., via hashing into a group [BLS01], and needs to be appropriately reflected in idealized models, particularly in the context of knowledge extraction.

For protocol designers relying on $\mathcal{G}\text{-}\mathtt{oGG}$, we provide a series of lemmas (Section 4) that simplify the process of writing proofs by enabling a transition to a *symbolic* functionality. Symbolic treatment of group exponents is a technique that is at the core of many GGM proofs.

Using our security proof framework, we prove (Section 5) that Groth16 UC-realizes the weak ideal functionality $\mathcal{F}$-wNIZK in the presence of $\mathcal{G}$-oGG.[7] We stress that Groth16 is proven secure as-is.

In particular, we achieve UC security without the overhead associated with UC SNARK compilers (e.g., [KZM+15, ARS20, BS21, LR22b, CSW22, AGRS23, GKO+23]). To the best of our knowledge, (simulation) extractability of Groth16 has been concretely analyzed only in the AGM [FKL18, BKSV21], but not in the GGM. Along the way, our analysis (Theorem 1) explicitly provides a concrete upper-bound on the distinguishing advantage of any environment, depending on its query complexity, the size of the group, and the size of the circuit, as well as the simulator query complexity.

Finally, we propose a way (Section 6) to deal with composition of protocols sharing a generic group in cases where *some* protocols cannot tolerate their group operations being observed.

1.2 Overview of our techniques

The restricted observable global generic group functionality. Observability of generic group operations should be sufficiently broad to allow a UC simulator to extract useful information from the adversary and environment, but it should not allow the environment to learn secret-dependent operations performed by honest parties. This tension goes to the core of compositional proofs: we need to strike a balance between information available to the security proof (UC simulator) for one protocol in a way that does not reveal too much about *other* protocols (UC environment) that would impact *their* security proofs. For random oracles $\mathcal{G}$-roRO, where observability is also used, this balance is easy to achieve via domain separation[8]: hashes of (sid, x) belong to session sid, and they become observable if computed in some session $sid' \neq sid$.

While domain separation for random oracles is easily modeled, designing the right domain separation mechanism for generic groups is far less obvious. A natural idea is to implement domain separation for groups via session-specific group generators, by assigning session sid a random generator g_{sid}. Intuitively, all operations done on g_{sid} or group elements derived from it belong to session sid. Operating on elements from a foreign session is deemed "illegal" and such operations are observable. However, compared to random oracles, there are additional difficulties: one can take two group elements g_{sid} and $g_{sid'}$ in two different sessions and meaningfully operate on them. This raises the question of whether cross-session operations such as $g_{sid} + g_{sid'}$ are observable, which session the resulting group element belongs to, and how we keep track of the sessions each group element belongs to.

Roughly speaking, in our approach, $\mathcal{G}$-oGG keeps track of the components of a group element in a symbolic way. Every generator g_{sid} corresponds to a formal (polynomial) variable X_{sid}. A group element such as $g_{sid} + g_{sid'}$ is associated with the polynomial $\mathsf{X}_{sid} + \mathsf{X}_{sid'}$. A group operation in protocol session sid is *illegal* (and hence observable) if the polynomial associated to the operation's result contains any foreign-session variables $\mathsf{X}_{sid'}$ (or a constant term). In other words, operations that involve other sessions' generators (as kept track of via polynomials) are observable. The formalization with polynomials avoids subtle issues with simpler approaches (Appendix D), where an element computed as $g_{sid} + g_{sid'} - g_{sid'}$ is

[7] Here *weak* refers to the fact that proofs may be re-randomizable, but are otherwise non-malleable. As observed by Kosba et al. [KZM+15, KMS+16] this weak version suffices in typical applications. As an analogy, many use cases of signatures only require existential unforgeability rather than full-fledged strong unforgeability.

[8] For the reader unfamiliar with domain separation approaches for global UC functionalities, Appendix C offers an explanation.

incorrectly associated with both sessions sid and sid', which causes issues with either too much or too little observability.

In the explanation above, every session sid only has a single generator g_{sid}. In our final formulation of $\mathcal{G}$-oGG (Section 3), a protocol can simply also call a Touch interface on any group element not already belonging to other sessions to declare it as an additional generator for its session. Hence every session can have multiple generators $g_{sid,1}, g_{sid,2}, \ldots$ and the observability mechanism generalizes naturally (the explanation above applies verbatim to the multiple-generator setting).

Cross-session element reuse. Note that in the $\mathcal{G}$-oGG model, the environment/adversary is not restricted in the way it can use group elements. In contrast to the UC-AGM, we allow the environment/adversary to take a group element output by some protocol, and use it to attack another protocol without any restriction. The crucial difference is how the knowledge of discrete logarithms is managed in UC-AGM vs. $\mathcal{G}$-oGG. In the UC-AGM, providing knowledge of discrete logarithms is the task of the environment/adversary. This is unfortunate because we also need to hide certain discrete-logarithm representations from the environment/adversary, e.g., as part of a simulation strategy. Additionally, different protocols have different AGM representation bases, and the environment is typically not able to convert a representation from one basis to another. In the UC-AGM, this leads to the adversary being effectively forbidden to use foreign group elements to attack another protocol.

With $\mathcal{G}$-oGG, there is no burden on the environment/adversary to keep track of representations. The knowledge of discrete-logarithm representations is effectively maintained by $\mathcal{G}$-oGG through observations: certain group operations are observable, and from those observations, anyone can compute (partial[9]) discrete-logarithm representations. As a consequence, the environment/adversary *is* allowed to take group elements from one session and use them to attack another session. The only "restriction" here is that group operations on foreign group elements are *observable*. That "restriction" makes it so protocols have to contend with observability, which makes it harder to prove constructions secure. It does not unnaturally impact the ability of the adversary to execute a wide range of real-world attacks.

Hashing and oblivious sampling. The encodings of group elements in our $\mathcal{G}$-oGG functionality belong to fixed sets that are of the same size as the group order. This is a closer modeling of how groups are used in practice (compared to, say, random encoding sets, where one does not even know in advance which of the encodings actually correspond to group elements). Crucially, this choice also allows adversaries and protocols to sample group elements in arbitrary ways, and thus allows us to avoid explicit modeling of oblivious sampling or hashing. (Such modeling is introduced for AGM in [LPS23, BFHK23], though to the best of our knowledge not yet ported to UC-AGM.) Fixing the sets of valid group encodings also allows hashing into groups via an independent (possibly global) random oracle functionality in parallel to a group functionality. (And whether or not this hashing is extractable or programmable is left to that functionality [CDG+18].) Conveniently, this means that we do not have to explicitly model a "hash-into-group" interface for generic groups: this functionality can be emulated using an external random oracle hashing into the set of valid group encodings.

Embedding generic groups into UC. Technically speaking, our $\mathcal{G}$-oGG is simply a *standard* UC functionality. It is *global*, meaning that instead of being a subroutine to a single protocol session, it accepts queries from *all* protocols as well as the environment in arbitrary sessions. For the notion of composability in the presence of global functionalities such as $\mathcal{G}$-oGG, we refer to the UCGS (UC with global subroutines) framework of [BCH+20], whose

[9] "Partial" in the sense that observations are sufficient for the simulator of session sid to learn the parts of the representation that pertain to the generators of sid; see Section 4.3 for the details.

composition theorem shows how to use the *original* UC composition theorem in the presence of global functionalities. (This work also points out certain gaps and shortcomings with the traditional GUC framework [CDPW07].) One of the advantages of modeling generic groups as a standard UC global functionality $\mathcal{G}$-oGG is that we do not require any modifications to the UC framework (we simply refer to the UCGS composition theorem for composition in the presence of $\mathcal{G}$-oGG). This is in contrast to other modeling approaches, such as the UC-AGM.

UC-SNARKs without overhead. Observable global generic groups are a practical means to study the UC security of efficient constructions. As a concrete application of relevance, we show that the Groth16 SNARK, without any modifications, UC-realizes the weak NIZK functionality $\mathcal{F}$-wNIZK in the $\mathcal{F}$-CRS-hybrid model and in the presence of $\mathcal{G}$-oGG (Theorem 1). To the best of our knowledge, this is the first result to establish the UC security of Groth16 with zero overhead.

Following [KZM$^+$15, KMS$^+$16], our goal is to UC-realize a slightly relaxed NIZK functionality which allows an adversary to maul an existing proof string π into a new one π^* but for the same statement x. This relaxation is necessary for Groth16 as its proof string can be re-randomized to obtain another valid proof [GM17]. Crucially, it still remains hard to obtain forged proof π^* for a new statement $x^* \neq x$. We analyze Groth16 as a canonical example due to its popularity in a number of deployed systems, and we believe our analysis should extend to its non-rerandomizable variants such as Groth–Maller [GM17] and Bowe–Gabizon [BG18] to show they UC-realize the strong NIZK functionality.

As part of our analysis, we introduce a set of technical lemmas, which provide a reusable template for formal analyses in the presence of global groups. These lemmas essentially allow one to operate with respect to a cleaner global functionality $\mathcal{G}$-oSG that is purely symbolic. In effect, they allow using the Schwartz–Zippel lemma (and in particular extraction of representations of group elements) in the UC setting. In more detail, we introduce a "fully symbolic" counterpart of the aforementioned $\mathcal{G}$-oGG, where every encoded group element maps to a formal polynomial instead of a $\mathbb{Z}_p$ element. In this way, one can guarantee perfect domain separation by ruling out exceptional events in which two group operations occurring in different sessions accidentally output the same group element. Our general lemma shows that one can switch to a hybrid UC experiment in the presence of the symbolic generic group functionality $\mathcal{G}$-oSG accepting a negligible loss in security.

Moreover, we provide a lemma that introduces a routine which makes a given *simulator* fully symbolic as well. Typically, a simulator for UC-NIZK uses secret random exponents (known as simulation trapdoor) to simulate the CRS and proof strings. After invoking this lemma, one can treat these random exponents as formal variables. We then apply these lemmas to analyze UC security of Groth16. The combination of our technical lemmas allows for clean and modular analysis of Groth16 in the UC setting. In particular, once we view all the random exponents in the current session as formal variables, we can reuse the existing weak simulation-extractability analysis of Groth16 [BKSV21] almost as is.

Composition when unobservability is required. The issue with using group elements from one protocol to attack another (as described above) in the UC-AGM is not unnatural, but rather points to an inherent conflict for composability in algebraic/generic group settings. $\mathcal{G}$-oGG tackles this issue not by restricting the environment (and hence the space of allowed attacks), but by making security proofs harder, essentially erring on the safe side. It does not, on its own, solve the inherent conflict. The observation rules of $\mathcal{G}$-oGG are well-suited for applications that can largely follow domain separation, such as SNARKs, where the prover only operates on CRS elements. However, in other protocols, when a party applies a secret to group elements *not* necessarily in its session, those operations are observable and the secret is effectively leaked. For example, a party in the ElGamal encryption scheme[10] would receive a ciphertext (c_1, c_2) from the environment and compute the plaintext $c_2 - sk \cdot c_1$. If the environment supplies c_1 that does not belong to the ElGamal protocol's session (e.g., a Groth16 CRS element), then the operation $sk \cdot c_1$ becomes observable, leaking the secret key to everyone. This is an inherent conflict with composition. The ElGamal protocol is interested in having unobservable operations on foreign elements. Conflicting with this, Groth16 *requires* that operations on its CRS by ElGamal are observable. Concretely, if decryption were afforded unobservability, then the decryption operation can effectively be used to compute a part of a valid Groth16 proof that the Groth16 UC simulator cannot trace, making extraction impossible.

We suggest a way to resolve this conflict by adapting a slight tweak to UC composition proofs. On a high level, when proving the composition of ElGamal and Groth16, one would *first* replace $\mathcal{F}$-wNIZK by the concrete Groth16 protocol. After that, observability is not needed anymore (as it is only used by the Groth16 simulator in the ideal world, not by the real-world protocol itself) and can be removed (conceptually). *Then*, one would replace $\mathcal{F}$-Enc by ElGamal. This replacement now happens in a setting where observation does not exist anymore. We sketch this approach in Section 6, but leave details for future work.

Painting the big picture, attacks involving cross-session use of group elements in the UC-AGM are partially disallowed, making it easy to prove a wide range of applications secure but restricting the class of covered attacks. Cross-session attacks are fully allowed with $\mathcal{G}$-oGG, meaning that we allow for all possible attacks, but such cross-session use results in observable operations, which rules out certain applications. However, this issue is mitigated with the approach described in Section 6. So overall, we get the best of both worlds: We can prove composition for a wide range of applications, in a model that does not restrict the environment.

Paper organization. The rest of the paper is organized as follows. Section 2 summarizes technical preliminaries. In Section 3, we formally introduce the restricted observable global generic group functionality $\mathcal{G}$-oGG. Section 4 states useful technical lemmas which provide a reusable template for formal analyses in the presence of global groups. In Section 5, we formally analyze UC security of the Groth16 SNARK in the presence of $\mathcal{G}$-oGG. Section 6 provides a tweak to UC composition proofs when unobservability is required. We conclude the paper with future work suggestions in Section 7. Apart from full proofs, the appendix also discusses additional related work in Appendix A, it answers frequently asked questions in Appendix B. Appendix C further discusses observability in global functionalities, while Appendix D discusses failed attempts for designs of $\mathcal{G}$-oGG, motivating design decisions.

1.3 Related work

Criticism and alternatives to the generic-group model. The generic-group model (GGM) is not without criticism. First, similar to random oracles, one can prove (artificial) schemes secure in the GGM that become provably insecure when instantiated with any concrete group [Den02]. Furthermore, applying the GGM in certain (non-generic) scenarios can lead to spurious security proofs [SPMS02].

In addition, the GGM only provides security guarantees against *generic* adversaries. However, we know that the fastest attacks on the discrete-logarithm problem in elliptic curve pairing groups make use of the specific structure of $\mathbb{G}_t$ via index calculus methods. As a result, the guarantees provided by the GGM are somewhat less meaningful. The semi-generic group model [JR10] addresses this weakness by modeling $\mathbb{G}_t$ as non-generic (while

[10] ElGamal is not a UC-secure encryption scheme. We are using it here for the sake of simplicity of illustration. The same principle applies to CCA2 secure variants of ElGamal, such as Cramer-Shoup [CS98].

$\mathbb{G}_1, \mathbb{G}_2$ are still generic groups). In practice, even with index-calculus methods, breaking the discrete-logarithm assumption (or any reasonable related assumption) is infeasible. So while there is some speed-up between the generic and non-generic attackers, the speed-up is not meaningful for suitably chosen pairing groups.

Finally, obliviously sampling a group element (or hashing into the group) is a widely used feature, which is often not supported by the GGM, causing issues [LPS23, BFHK23]. The generic-group modeling in our paper enables oblivious sampling as discussed above.

Overall, while there is criticism of the generic-group model, it is still widely used as a useful tool to establish security guarantees in the absence of stronger formal evidence.

The algebraic-group model (AGM) [FKL18] was born out of criticism of the GGM. Security in the AGM is established with respect to a restricted class of *algebraic* adversaries, which are required to always supply the (discrete-log) representations of their output group elements in terms of the input elements that they have seen so far. This means that intuitively, because an AGM adversary gets to see proper group element encodings rather than random ones, the AGM is a weaker (less severely restricting) model than the GGM (though depending on the exact AGM/GGM formalization, this intuition is not necessarily formally true [ZZK22]). The AGM does not support oblivious hashing, but can be extended to do so [LPS23].

The UC-AGM [ABK+21] excludes cross-session group element attacks, as explained above. For this reason, despite the AGM *usually* being the better model than the GGM, the same does not seem to hold true when it comes to questions of composability.

UC-secure proof systems. Although a number of papers study generic transformations that lift NIZK proof systems in the stand-alone setting into a UC-secure one [KZM+15, ARS20, BS21, LR22b, CSW22, AGRS23, GKO+23], they end up with proof sizes that are linear in the witness size, sacrificing succinctness, or else introduce significant overheads in the proving time. To realize the ideal functionality, these UC-lifting compilers typically output a proof system satisfying the simulation-extractability property [Sah99, DDO+01, Gro06, FKMV12]. While Groth16 and its variants already have proof of simulation-extractability in the GGM/AGM [BKSV21, BG18, GM17], their implications to composable security have been unclear prior to our work. So far, there is little work on SNARKs being UC-secure as is, i.e., without having to apply a transformation, which is the state of the art. The exception to this is a recent concurrent work [CF24] that proves Micali's SNARK [Mic00] and certain IOP-based SNARKs obtained via the BCS transform [BCS16] proven UC-secure in the presence of $\mathcal{G}$-$\texttt{rpoRO}$, i.e., in the random-oracle setting. We defer a more complete review of UC-secure proof systems to Appendix A.

2 Preliminaries

2.1 Notation

Functions and pseudocode. For a (partial) function $\tau : A \to B$, define the image $\mathrm{im}(\tau) = \{y \mid \exists x : \ \tau(x) = y\} \subseteq B$ and the domain $\mathrm{dom}(\tau) = \{x \mid \tau(x) \neq \bot\} \subseteq A$. We write "assert ϕ" as a shorthand for "if $\neg\phi$, then return $\bot$". List concatenation is denoted by colon $(A : B)$.

Sets and polynomials. For subsets $A, B \subseteq R$ of a ring R, $r \in R$, define $A+B := \{a+b \mid a \in A, b \in B\}$, $r \cdot A := \{r \cdot a \mid a \in A\}$, and $A \cdot B := \{a \cdot b \mid a \in A, b \in B\}$. Still, $A^n = A \times A \times \cdots \times A$ denotes the n-fold Cartesian product.

We denote scalars by lower-case letters (e.g., $a \in \mathbb{Z}_p$), and formal variables/polynomials in sans-serif font (e.g., $\mathsf{A} \in \mathbb{Z}_p[\mathsf{X}]$). We also consider polynomials and variable with negative degree, e.g., $2\mathsf{X} + 3\mathsf{X}^{-1} \in \mathbb{Z}_p[\mathsf{X}, \mathsf{X}^{-1}]$. Sets or maps involving scalars are generally written as

S, and if they involve polynomials, they are written as S. For Var, a set of variables, we let $\mathsf{Var}^{\pm 1} := \mathsf{Var} \cup \mathsf{Var}^{-1}$, where Var^{-1} is a set containing the inversion of variables in Var.

Let R be a ring of polynomials, $\mathsf{A}, \mathsf{B} \in R$, and $\mathsf{L} \subseteq \mathsf{R}$ be a finite list of ring elements. Then $\langle \mathsf{L} \rangle_{\mathsf{R}} := \sum_{\mathsf{x} \in \mathsf{L}} \mathsf{x} \cdot \mathsf{R} \subseteq \mathsf{R}$ is the ideal generated by the elements of L. For example, for $\mathsf{R} = \mathbb{Z}_p[\mathsf{X}, \mathsf{X}', \mathsf{Y}]$, we have that $\langle \mathsf{X}, \mathsf{X}', \mathsf{Y} \rangle_{\mathbb{Z}_p[\mathsf{X},\mathsf{X}',\mathsf{Y}]}$ is the set of all polynomials with no constant term and $\langle \mathsf{X}, \mathsf{X}' \rangle_{\mathbb{Z}_p[\mathsf{X},\mathsf{X}',\mathsf{Y}]}$ is the set of all polynomials only containing non-constant monomials in X or X' (e.g., $2\mathsf{X} + 3\mathsf{X}' + 4\mathsf{X}\mathsf{X}' + 5\mathsf{X}\mathsf{Y} \in \langle \mathsf{X}, \mathsf{X}' \rangle_{\mathbb{Z}_p[\mathsf{X},\mathsf{X}',\mathsf{Y}]}$, but $\mathsf{Y}, \mathsf{X} + 3 \notin \langle \mathsf{X}, \mathsf{X}' \rangle_{\mathbb{Z}_p[\mathsf{X},\mathsf{X}',\mathsf{Y}]}$).

We say that $\mathsf{a} = \mathsf{b} \mod \langle \mathsf{L} \rangle_{\mathsf{R}}$ (or simply, $\mathsf{a} = \mathsf{b} \mod \mathsf{L}$) if $\mathsf{a} - \mathsf{b} \in \langle \mathsf{L} \rangle_{\mathsf{R}}$. For example, $\mathsf{X} + 5\mathsf{Y} + 7\mathsf{X}\mathsf{Y} + 3 = \mathsf{X} + 3 \mod \mathsf{Y}$.

Lemma 1 (Schwartz–Zippel). *Let $\mathbb{F}$ be a finite field of order $p > 1$, $\mathsf{Var} = (\mathsf{X}_1, \ldots, \mathsf{X}_n)$ be a list of formal variables and $\mathsf{f} \in \mathbb{F}[\mathsf{Var}]$ be a polynomial with $\mathsf{f} \neq 0$. Then*

$$\Pr[\mathsf{f}(x_1, \ldots, x_n) = 0] \leq \deg(\mathsf{f})/p \ ,$$

where the probability is over $x_1, \ldots, x_n \xleftarrow{\$} \mathbb{F}$.

Lemma 2 (Schwartz–Zippel for Laurent polynomials). *Let $\mathbb{F}$ be a finite field of order $p > 1$, $\mathsf{Var} = (\mathsf{Y}_1, \ldots, \mathsf{Y}_n)$ be a list of formal variables and $\mathsf{f} \in \mathbb{F}[\mathsf{Var}^{\pm 1}]$ be a Laurent polynomial with $\mathsf{f} \neq 0$. Then*

$$\Pr[\mathsf{f}(y_1, \ldots, y_n) = 0] \leq 2\deg(\mathsf{f})/(p-1) \ ,$$

where the degree of a Laurent polynomial is defined as the maximal absolute value of the exponent of any term, and the probability is over $y_1, \ldots, y_n \xleftarrow{\$} \mathbb{F}^*$.

2.2 Generic bilinear groups

Philosophically, the generic-group model represents an idealization of a (potentially bilinear) group, where protocols and attackers can only (meaningfully) interact with the group by executing group operations. They cannot exploit any additional structure of the group. The generic-group model has been formulated in two majors forms: one due to Shoup and Nachaev [Nec94, Sho97] that idealizes element encodings as random strings, and the other due to Maurer [Mau05] that treats group elements as abstract handles. (See, e.g., [Zha22] for a modern perspective and comparisons.) In this work we focus on Shoup and Nachaev's model, adopted to the case of bilinear groups.

The bilinear generic-group model is parameterized by $(p, S_1, S_2, S_{\mathrm{t}})$, consisting of two (carrier) sets of size p corresponding to source groups S_1 and S_2, and another, also of size p, corresponding to the target group S_{t}. All parties, honest or otherwise, are given oracle access to three random injections $\tau_i \xleftarrow{\$} \mathrm{Inj}(\mathbb{Z}_p, S_i)$ for $i = 1, 2, \mathrm{t}$ as well as $(\tau_1(1), \tau_2(1), \tau_{\mathrm{t}}(1))$.

In this model, parties also get oracle access to three compatibly defined group operation oracles which invert a given element via τ_i^{-1}, perform addition over $\mathbb{Z}_p$, and re-encode via τ_i. Finally, a pairing operation allows "multiplying" two elements, one in S_1 and the other in S_2, via respective inversions under τ_1 and τ_2, multiplication over $\mathbb{Z}_p$, and re-encoding via τ_{t}.

There are three prominent types of bilinear groups that are commonly used in practice, corresponding to whether the groups are different or if there is an isomorphism between the groups. From a generic-group perspective, in type-I groups $S_1 = S_2$ and their corresponding injections τ_1 and τ_2 are also identified. In type-II and type-III groups the injections remain independent, though for type-II groups one also provides oracle access to an isomorphism from the second source group to the first, implemented via inversion under τ_2^{-1} and re-encoding under τ_1. Here we focus on type-III bilinear groups with no isomorphism in either direction as these are most commonly used in practice. Throughout, we use additive notations for operations performed in all three groups.

A final distinction made in use of generic groups is whether (honest) group operations are performed with respect to the given set of "canonical" generators $(\tau_1(1), \tau_2(1), \tau_t(1))$ or whether random generators are used. This choice has security implications as shown in [BMZ19]. As we shall see, for our UC security proofs, it is critical that protocols use random generators.

2.3 The UC framework and its execution model

We rely on the Universal Composability (UC) framework [Can01]. However, our results could also be expressed using the concepts of other comparable frameworks [Mau10, Küs06, HS15, BDF+18]. Historically, the treatment of global resources required a more general and complex compositional framework [CR03, CDPW07]. Badertscher et al. [BCH+20] show how to view global functionalities as *global subroutines*, a concept that can be made precise within the latest installment of the plain UC framework [Can20b]. Here, we provide a summary of [Can20b] and refer interested readers to the original works for further details.

Formalism. In the UC framework, protocols are modeled as a system of Interactive Turing Machines (ITM). While an ITM itself is just a static piece of code, for each *session identifier* $sid \in \mathbb{N}$, we consider a collection of ITM instances (ITI) sharing the same *sid*. Each ITI is an instance of some ITM for a specific session and together they form the runtime notion of a protocol session. Each ITI in a given protocol session is also called a *party*.

The execution of a protocol Π involves a set of parties $\mathcal{P}$, the environment $\mathcal{Z}$ (which essentially behaves like an interactive distinguisher), and the *adversary* $\mathcal{A}$. The environment controls the flow of execution by interacting with the *adversary* $\mathcal{A}$ and choosing inputs to the parties involved in Π and receiving their outputs. An *identity bound* ξ places restrictions on whom $\mathcal{Z}$ can provide input to (e.g., to ensure the environment cannot make calls to subroutines of Π on behalf of Π). The execution terminates when the environment finally terminates with an output 0 or 1.

During an execution of Π, the adversary $\mathcal{A}$ may *corrupt* a subset of parties as defined by the security model in order to learn their internal states and gain control over these parties. In this paper, we focus on static corruption meaning that $\mathcal{A}$ chooses which party to be corrupted in the beginning of the execution.

We denote by $\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$ the distribution of a binary output by $\mathcal{Z}$ after an execution of Π in the presence of $\mathcal{A}$, where $\lambda \in \mathbb{N}$ is a security parameter, $z \in \{0,1\}^*$ is an auxiliary input to $\mathcal{Z}$, and the randomness for all ITMs are assumed to be sampled uniformly at random. We define the family (or ensemble) of random variables $\{\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$.

Recall that two binary distribution families X, Y indexed by $\lambda \in \mathbb{N}$, and $z \in \{0,1\}^*$ are called *indistinguishable* (denoted $X \approx Y$) if for all $c, d \in \mathbb{N}$, there exists a $\lambda_0 \in \mathbb{N}$ such that for all $\lambda > \lambda_0$ and all $z \in \cup_{\kappa\le\lambda^d}\{0,1\}^\kappa$, $|\Pr[X(\lambda, z) = 1 - \Pr[Y(\lambda, z) = 1]| < \lambda^{-c}$.

UC Security. Intuitively, we consider that a protocol Π in the presence of an adversary $\mathcal{A}$ successfully UC-emulates another (typically more idealized) protocol Φ if there exists another adversary (aka. *simulator*) $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish the execution of Φ with $\mathcal{S}$ from that of Π with $\mathcal{A}$.

Definition 1 (UC emulation). *A protocol Π is said to* UC-emulate *Φ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for all PPT environment $\mathcal{Z}$*

$$\{\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} \approx \{\mathsf{EXEC}_{\Phi,\mathcal{S},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} .$$

To define the security of protocol $\mathit{\Pi}$ in the UC framework, one describes an ideal functionality $\mathcal{F}$ which captures the desired functionality of the task in hand in the form of an ITM. One then defines $\mathit{\Pi}$ UC-secure if $\mathit{\Pi}$ UC-emulates the *ideal protocol* $\mathit{\Phi} = \mathsf{IDEAL}_{\mathcal{F}}$. The ideal protocol $\mathsf{IDEAL}_{\mathcal{F}}$ models an idealized run of protocol execution: the *simulator* $\mathcal{S}$ only interacts with $\mathcal{Z}$ and influences the execution through the prescribed interfaces of $\mathcal{F}$, and the parties $\mathcal{P}$ are replaced with the so-called *dummy parties* $\tilde{\mathcal{P}}$ which merely forward the inputs from $\mathcal{Z}$ to $\mathcal{F}$ and the responses back from $\mathcal{F}$ to $\mathcal{Z}$.

Syntax for ideal functionalities and protocols. In this paper, we use the following syntax to enable more precise (code-based) specifications of ideal functionalities. We describe $\mathcal{F}$ as a collection of *internal states* and *interfaces*. As usual, upon the first invocation of $\mathcal{F}$ within session *sid* its instance gets created with initial internal states. We model this routine by introducing $\mathcal{F}.\textsc{Init}_{\mathit{sid}}()$, which can be called only once. Once an instance of $\mathcal{F}$ is created within *sid*, the subsequent calls to $\textsc{Init}_{\mathit{sid}}()$ are ignored. If $\mathcal{F}$ comes with interface $\textsc{Interface}$, the (co-)routine "$\mathcal{F}.\textsc{Interface}_{\mathit{sid}}(\mathsf{in})$" defines the behavior of the interface for session *sid* on input in, and returns the resulting output, potentially after interacting with the simulator. Every invocation of $\textsc{Interface}_{\mathit{sid}}$ may update the internal state of an instance of $\mathcal{F}$.

UC with global functionalities. [BCH+20] model *global functionalities* within the basic UC framework described above. Unlike a (local) functionality $\mathcal{F}$, a single instance of a global functionality $\mathcal{G}$ may take input from and provide outputs to multiple instances of protocols and local functionalities. Moreover, the environment $\mathcal{Z}$ can directly interact with $\mathcal{G}$ without going through spawned instances of the adversary. The definition of security can be naturally extended in the presence of a global functionality as we define next.

Definition 2 (UC emulation with global setup). *Let $\mathcal{G}$ be a global functionality. A protocol $\mathit{\Pi}$ is said to* UC-emulate $\mathit{\Phi}$ in the presence of $\mathcal{G}$, *if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for all PPT environment $\mathcal{Z}$,*

$$\{\mathsf{EXEC}_{\mathit{\Pi},\mathcal{G},\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} \approx \{\mathsf{EXEC}_{\mathit{\Phi},\mathcal{G},\mathcal{S},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} .$$

Here, $\mathsf{EXEC}_{\mathit{\Pi},\mathcal{G},\mathcal{A},\mathcal{Z}}(\lambda, z)$ is defined in terms of $\mathsf{EXEC}_{\mu[\mathit{\Pi},\mathcal{G}],\mathcal{A},\mathcal{Z}}(\lambda, z)$, where the so-called management protocol μ allows $\mathit{\Pi}$ to interact with $\mathcal{G}$ but additionally grants access to $\mathcal{G}$ to $\mathcal{Z}$.

In [BCH+20], the authors present a *composition theorem for global subroutines (UCGS theorem)*, which states the following: if a protocol $\mathit{\Pi}$ UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}$, then the protocol $\rho^{\mathit{\Pi},\mathcal{G}}$ that is identical to $\rho^{\mathcal{F},\mathcal{G}}$ except that all instances of the ideal functionality $\mathcal{F}$ are replaced by instances of the real protocol $\mathit{\Pi}$, UC-emulates $\rho^{\mathcal{F},\mathcal{G}}$ in the presence of $\mathcal{G}$.

2.4 Weak NIZK functionality

In Functionality 1 we formalize $\mathcal{F}$-`wNIZK`, the *weak* NIZK ideal functionality that we will be realizing. $\mathcal{F}$-`wNIZK` is parameterized by polynomial-time relation $\mathcal{R}$, and runs with parties $\mathcal{P}$ and an ideal process adversary $\mathcal{S}$. It stores a proof table T which is initially empty. "Weak" refers to the fact that proofs may be malleable. Our formalization slightly differs from [KZM+15, Figure 3] in that mauling of proofs is performed by the simulator and not via an explicit maul interface. We note that with Line 9 removed, we obtain an ideal functionality for a standard ("strong") NIZK.

Here we consider the case of static corruption. This is sufficiently strong to also give adaptive corruption for $\mathcal{F}$-`wNIZK` (where the all queried (x, w) are returned upon corruption) assuming secure erasure (of randomness). In order to have a simpler functionality, we do not

Functionality 1: $\mathcal{F}$-wNIZK

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

$\underline{\textsc{Prove}_{sid}(x, w)}$
2: if $(x, w) \notin \mathcal{R}$ then return $\bot$
3: $\pi \leftarrow \mathcal{S}.\textsc{Simulate}_{sid}(x)$
4: $T \leftarrow T \cup (x, \pi)$
5: return $\pi$

$\underline{\textsc{Verify}_{sid}(x, \pi)}$
6: if $(x, \pi) \in T$ then return 1
7: $w \leftarrow \mathcal{S}.\textsc{Extract}_{sid}(x, \pi)$
8: if $(x, w) \in \mathcal{R}$ then $T \leftarrow T \cup (x, \pi)$
9: if $(w = \mathsf{maul} \wedge (x, *) \in T)$ then $T \leftarrow T \cup (x, \pi)$
10: if $(x, \pi) \in T$ then
11:     return 1
12: else
13:     return 0
```

Functionality 2: $\mathcal{G}$-GG

```
$\underline{\textsc{Init}()}$
1: for $i \in \{1, 2, \mathrm{t}\}$ do
2:     $\tau_i \xleftarrow{\$} \mathrm{Inj}(\mathbb{Z}_p, S_i)$

$\underline{\textsc{Op}_{sid}(i, g_1, g_2, a_1, a_2)}$
3: assert $(g_1, g_2, a_1, a_2) \in S_i^2 \times \mathbb{Z}_p^2$
4: $h \leftarrow \tau_i(a_1 \tau_i^{-1}(g_1) + a_2 \tau_i^{-1}(g_2))$
5: return $h$

$\underline{\textsc{CanonicalGen}_{sid}(i)}$
6: return $\tau_i(1)$

$\underline{\textsc{Pair}_{sid}(g_1, g_2)}$
7: assert $(g_1, g_2) \in S_1 \times S_2$
8: $h \leftarrow \tau_{\mathrm{t}}(\tau_1^{-1}(g_1) \cdot \tau_2^{-1}(g_2))$
9: return $h$
```

model that a previously invalid proof must not subsequently become valid. Note, however, that Groth16 enjoys full consistency.

3 The global observable generic group functionality

In this section, we first go over the (strict) global generic group model as a warm-up, and then introduce the restricted observable global generic group model, which is what we are going to use to prove UC security of Groth16.

3.1 Warm-up: The (strict) global generic group functionality

We focus on type-3 bilinear groups and Shoup's style of generic groups with random encodings (cf. Section 2.2). We can easily model such (unobservable) generic bilinear groups as a (global) UC functionality $\mathcal{G}$-GG as in Functionality 2 (similar to, for example, [CNPR22]). As in standard generic type-III bilinear groups, $\mathcal{G}$-GG is parameterized by a prime p and three sets S_i for $i \in \{1, 2, \mathrm{t}\}$ each of size p. $\mathcal{G}$-GG starts by initializing three random injections $\tau_i : \mathbb{Z}_p \to S_i$ for $i \in \{1, 2, \mathrm{t}\}$. (This choice can be made efficient in the standard way, via lazy sampling.)

The functionality $\mathcal{G}$-GG offers three interfaces to protocols. They can use $\mathcal{G}$-GG to access the "canonical" generators $\tau_i(1)$ via CanonicalGen. As with standard generic groups, $\mathcal{G}$-GG also offers an Op and a Pair interface. We slightly extend Op to compute an arbitrary linear operation $a_1 \cdot g_1 + a_2 \cdot g_2$ (rather than just $g_1 + g_2$). This is without loss of generality and is used to spare algorithms from implementing double and add.

Because the sets S_i are public and of size p, protocols (and adversaries) can (obliviously) sample group elements of their choice. This could be via an arbitrary algorithm that has an unspecified output distribution. (Some formalizations allow S_i to be a much larger set than $\mathbb{Z}_p$, which prevents these powers.) Moreover this choice better conforms to practical groups (where the carrier sets of a bilinear group are fixed and publicly known).

This feature, when combined with an external random oracle functionality, also enables hashing into the group via random oracle. For this reason, and in contrast to, say, [CNPR22], we do not explicitly model a "hash-into-group" interface

As such, $\mathcal{G}$-`GG` can be seen as the generic-group equivalent of the "strict" global *random-oracle* functionality [CDG+18]. It can be used, for example, to analyze the UC security algebraic schemes like ElGamal when they share a generic group.

3.2 The (restricted) observable global generic group functionality

The ability to observe generic-group (and random-oracle) queries forms the basis of many proofs in cryptography. Functionally, $\mathcal{G}$-`GG` as defined has limited applicability, because it does not offer the UC simulator any "cheating power". This is in contrast to a local group [CNPR22] where the simulator takes over the group.

To enable applications where simulators need to observe queries made to the group, we augment $\mathcal{G}$-`GG` with *observation* capabilities. As seen in the analogous restricted observable global *random oracle* (e.g., [CDG+18]), these observation capabilities need to be appropriately *restricted* so as to not render all applications insecure.

Our global *restricted observable generic group* functionality $\mathcal{G}$-`oGG` is defined in Functionality 3. It contains all interfaces of $\mathcal{G}$-`GG`, together with two additional ones, Observe and Touch. If Observe and Touch are never called, then $\mathcal{G}$-`oGG` behaves identically to $\mathcal{G}$-`GG`. In particular, $\mathcal{G}$-`oGG`.$\textsc{Op}_{sid}(i, g_1, g_2, a_1, a_2)$ still effectively returns $h = \tau_i(a_1\tau_i^{-1}(g_1) + a_2\tau_i^{-1}(g_2))$, and the only difference to its counterpart in $\mathcal{G}$-`GG` is that the operation additionally keeps track of the way group elements are computed, which we discuss below. Similarly, $\mathcal{G}$-`oGG`.Pair differs from $\mathcal{G}$-`GG`.Pair only in maintaining some additional bookkeeping.

Our strategy to restrict observability is similar to (restricted) observable random oracles [CDG+18] in that we deploy a form of "domain separation".[11] $\mathcal{G}$-`oGG` introduces a notion of group elements belonging to certain sessions, which informs the observation rules. This notion, however, is somewhat nontrivial—after all, the entire group is shared equally among all sessions, with no algebraic differentiation between any two group elements. To associate group elements with sessions, we keep track of *polynomial representations* of group elements with respect to certain generators.

Generators. To start, protocols can claim (random) generators $g \in S_i$ for each group in their session by simply calling the $\textsc{Touch}_{sid}(i, g)$ procedure. Reminiscent of the Unix `touch` command, if g is already in use, nothing happens. Otherwise, g becomes a generator of the caller's session *sid*. (Protocols can choose g randomly to ensure that g is unused with overwhelming probability.) A formal variable X is associated with every touched generator g. The functionality keeps track of each session's generators in terms of their formal variables using lists $\mathsf{Var}_{i,sid}$ (to which X is appended). The canonical generators $\tau_i(1)$ do not belong to any particular session. Looking slightly ahead, every group element $h \in S_i$ will be associated with a (set of) polynomials $\mathsf{R}_i[h]$ that explain how the group element has been computed. For a touched generator g with associated formal variable X, the polynomial representation is simply $\mathsf{R}_i[h] = \{\mathsf{X}\}$. The canonical generators are represented with constant polynomials, $\mathsf{R}_i[\tau_i(1)] = \{1\}$.

Group operations. When executing group operations, $\mathcal{G}$-`oGG` keeps track of the polynomial representations corresponding to the resulting group element. Whenever two group elements are added, their polynomial representations are summed up to form the corresponding polynomial representation (Line 10 of Functionality 3). Whenever the pairing operation is applied, polynomial representations are multiplied (Line 22). For example, let g_1, g_2 be

[11] In Appendix C we give an overview of domain-separation approaches in global observable functionalities.

Functionality 3: $\mathcal{G}$-oGG

$\mathcal{G}$-oGG is (implicitly) parameterized with

- A prime number p
- Sets $S_1, S_2, S_t \subseteq \{0,1\}^*$ with $|S_i| = p$ for all $i \in \{1,2,t\}$.

$\mathcal{G}$-oGG maintains the following state:

- $\tau_i : \mathbb{Z}_p \to S_i$ three random encoding functions, mapping discrete logs $x \in \mathbb{Z}_p$ to their randomly encoded group elements $h \in S_i$.
- $\mathsf{Var}_{i,sid}$ initially empty lists of formal variables. // Keeps track of the group i formal variables belonging to session sid.
- $\mathsf{R}_i[h]$ for $i \in \{1,2,t\}, h \in S_i$ initially empty sets of polynomials // Keep track of polynomial representations corresponding to $h \in S_i$.
- Ob initially empty list of observable actions.

Furthermore, we use the following terms derived from the current state

- We write $\mathsf{Var}_{sid} = \mathsf{Var}_{1,sid} : \mathsf{Var}_{2,sid} : \mathsf{Var}_{t,sid}$ to refer to all variables of session sid (irrespective of which group).
- We write Var to refer to the concatenation of all Var_{sid} (i.e. over all sid).
- $\mathsf{Legal}_{sid} = \langle \mathsf{Var}_{sid} \rangle_{\mathbb{Z}_p[\mathsf{Var}_{sid}]} = \sum_{\mathsf{X} \in \mathsf{Var}_{sid}} \mathsf{X} \cdot \mathbb{Z}_p[\mathsf{Var}_{sid}]$. // Legal_{sid} is the set of polynomials that contain only this session's variables $\mathsf{X} \in \mathsf{Var}_{sid}$, and whose constant term is 0. For example, $15\mathsf{X}_{sid} + 7\mathsf{Y}_{sid} \in \mathsf{Legal}_{sid}$ and $3\mathsf{X}_{sid}\mathsf{Y}_{sid} \in \mathsf{Legal}_{sid}$, but $\mathsf{X}_{sid} + 3 \notin \mathsf{Legal}_{sid}$ and $\mathsf{X}_{sid} + \mathsf{X}_{sid'} \notin \mathsf{Legal}_{sid}$.

```
INIT()  // Invoked only upon creation
1: for $i \in \{1,2,t\}$ do
2:     $\tau_i \xleftarrow{\$} \mathrm{Inj}(\mathbb{Z}_p, S_i)$
3:     $\mathsf{R}_i[\tau_i(1)] \leftarrow \{1\}$

CANONICALGEN_{sid}(i)
4: return $\tau_i(1)$

OBSERVE_{sid}()
5: return $Ob$

OP_{sid}(i, g_1, g_2, a_1, a_2)
6: assert $(g_1, g_2, a_1, a_2) \in S_i^2 \times \mathbb{Z}_p^2$
7: for $j \in \{1,2\}$ do
8:     TOUCH_{sid}($i, g_j$)
9: $h \leftarrow \tau_i(a_1\tau_i^{-1}(g_1) + a_2\tau_i^{-1}(g_2))$
10: $\mathsf{R}_i[h] \leftarrow \mathsf{R}_i[h] \cup (a_1\mathsf{R}_i[g_1] + a_2\mathsf{R}_i[g_2])$
11: if $\exists \mathsf{f} \in \mathsf{R}_i[h] : \mathsf{f} \notin \mathsf{Legal}_{sid}$ then
12:     $Ob \leftarrow Ob : [(\text{OP}, i, g_1, g_2, a_1, a_2, h)]$
13: return $h$

TOUCH_{sid}(i, g)
14: if $\mathsf{R}_i[g] = \varnothing$ then
15:     Initialize fresh variable $\mathsf{X}$
16:     $\mathsf{Var}_{i,sid} \leftarrow \mathsf{Var}_{i,sid} : [\mathsf{X}]$
17:     $\mathsf{R}_i[g] \leftarrow \{\mathsf{X}\}$

PAIR_{sid}(g_1, g_2)
18: assert $(g_1, g_2) \in S_1 \times S_2$
19: for $i \in \{1,2\}$ do
20:     TOUCH_{sid}($i, g_i$)
21: $h \leftarrow \tau_t(\tau_1^{-1}(g_1) \cdot \tau_2^{-1}(g_2))$
22: $\mathsf{R}_t[h] \leftarrow \mathsf{R}_t[h] \cup (\mathsf{R}_1[g_1] \cdot \mathsf{R}_2[g_2])$
23: if $\exists \mathsf{f} \in \mathsf{R}_t[h] : \mathsf{f} \notin \mathsf{Legal}_{sid}$ then
24:     $Ob \leftarrow Ob : [(\text{PAIR}, t, g_1, g_2, h)]$
25: return $h$
```

generators associated with formal variables $\mathsf{R}_i[g_1] = \{\mathsf{X}_1\}, \mathsf{R}_i[g_2] = \{\mathsf{X}_2\}$. If we compute "$h = 1 \cdot g_1 + 3 \cdot g_2$", then the corresponding polynomial is $\mathsf{R}_i[h] = \{\mathsf{X}_1 + 3\mathsf{X}_2\}$. If we further compute "$h' = 2 \cdot h + 50 \cdot g_1$", then $\mathsf{R}_i[h'] = \{52\mathsf{X}_1 + 6\mathsf{X}_2\}$. Note that by design, polynomials in R_1 and R_2 are of degree 1 or 0, and polynomials in R_t for the target group are of degree at most 2.

It may happen that there are two polynomial representations $\mathsf{f} \neq \mathsf{f}'$ for the same group element h. For this reason, $\mathsf{R}_i[h]$ is formally modeled as a *set* containing all known representations. However, by Schwartz–Zippel (Lemma 1), for sufficiently large groups, $\mathsf{R}_i[h]$ will be a singleton set with overwhelming probability (we formally establish this in the UC setting in the proof of Lemma 3).

Observation rules. With the above bookkeeping mechanisms, we have polynomials $\mathsf{f} \in \mathsf{R}_i[h]$ associated to each group element h, and sessions to each polynomial variable $\mathsf{X} \in \mathsf{Var}_{sid}$. This now allows us to establish the observation rules. For this, we say that a group element h is *legal* in session *sid* if its associated polynomial(s) $\mathsf{f} \in \mathsf{R}_i[h]$ do not contain variables $\mathsf{X}_{sid'} \in \mathsf{Var}_{sid'}$ of foreign sessions $sid' \neq sid$ (and no constant terms, which correspond to the canonical generators). This the set Legal_{sid} in Functionality 3 formally defines the set of legal polynomials for session *sid*. A group operation or pairing operation is *observable* if its result is not legal in the caller's session. If an operation is observable, then the input to the operation is added to a global list *Ob* in Line 12 and 24. *Ob* can be read by anyone (environment, simulator, adversary, even, theoretically, protocol entities) by calling OBSERVE.

Intuitively, in order to not be observed, the protocol in session *sid* must only operate with group elements that were derived from its session's generators, with no involvement of generators from other sessions *sid'*. To comply with domain separation, protocols in session *sid* must only operate with group elements that were derived from their session's generators, with no involvement of generators from other sessions *sid'*. For example, if $\mathsf{R}_i[h] = \{4\mathsf{X}_{sid} + 3\mathsf{X}'_{sid}\}$, where $\mathsf{X}_{sid}, \mathsf{X}'_{sid} \in \mathsf{Var}_{i,sid}$ are associated with session *sid*, then clearly, h belongs to session *sid*. An $\textsc{Op}_{sid}$ operation called by a party in session *sid*, resulting in h is an example of an *unobservable* operation. However, if $\mathsf{R}_i[h] = \{4\mathsf{X}_{sid} + 3\mathsf{Y}_{sid'}\}$, where $\mathsf{Y}_{sid'} \in \mathsf{Var}_{i,sid'}$ belongs to session $sid' \neq sid$, then h does not belong to either session. An $\textsc{Op}_{sid}$ operation called by a party in session *sid* (or indeed any other session), resulting in h is an example of an *observable* operation. For a pairing operation $\textsc{Pair}_{sid}(h_1, h_2) = h$, we naturally get that if, say, $\mathsf{R}_1[h_1] = \{\mathsf{X}_{sid}\}$ and $\mathsf{R}_2[h_2] = \{3\mathsf{Z}_{sid}\}$, then for the result h, we get $\mathsf{R}_\mathrm{t}[h] = \{3\mathsf{X}_{sid}\mathsf{Z}_{sid}\}$, which indicates that h is legal (unobservable). If, however, instead $\mathsf{R}_2[h_2] = \{3\mathsf{Z}_{sid'}\}$ with $\mathsf{Z}_{sid'} \in \mathsf{Legal}_{2,sid'}$, then the result is illegal (hence observable), since $\mathsf{R}_\mathrm{t}[h] = \{3\mathsf{X}_{sid}\mathsf{Z}_{sid'}\} \not\subseteq \mathsf{Legal}_{sid}$.

Using $\mathcal{G}$-oGG in protocols. A protocol can set up its set of generators by sampling random group elements $g_1 \overset{\$}{\leftarrow} S_1, g_2 \overset{\$}{\leftarrow} S_2$, and TOUCHing them to make them part of the protocol's session. The protocol can then proceed naturally, performing group and pairing operations as usual. For example, Groth16 can choose a common reference string (CRS) based on g_1, g_2 (see Functionality 6).

With the observation rules in place, the simulator for session *sid* can be sure that it gets observation information pertaining to all group elements h whose polynomial $\mathsf{f} \in \mathsf{R}_i[h]$ involves any variable $\mathsf{X} \in \mathsf{Var}_{sid}$.

If the protocol stays within elements derived from its generators g_1, g_2 (e.g., the CRS and Groth16 proofs computed from it), those operations will, with overwhelming probability, not be observable. See Section 4 for a discussion on unlikely error events. A protocol *may* sometimes violate domain separation. For example, this is necessary in Groth16 when verifying a proof π received from the environment, which can potentially contain adversarially generated group elements belonging to other sessions. In this case, operations are observable, hence care must be taken that they do not leak any important information (which is not an issue for Groth16, as the verifier does not hold any secret information). We discuss handling protocols where this is an issue in Section 6.

Protocols can hash into the group (similarly to what we described in Section 3) by hashing into S_i (e.g., with a random oracle) and then TOUCHing the hash output. If there is sufficient entropy in the hashed element, it is likely that the hash output will belong to the hasher's session, making it safe to perform secret operations on it.

Canonical generators. The canonical generators g_1, g_2, g_t, available via CANONICALGEN correspond to discrete logarithms $\tau_i^{-1}(g_i) = 1$ and the constant polynomials $\mathsf{R}_i[g] = \{1\}$. In principle, they can be used by any protocol (session). However, operations involving the canonical generator will all be observable (any polynomial with a non-zero constant term is observable). This is a somewhat arbitrary choice, but makes for nicer algebraic properties of observability (e.g., the set $\mathsf{Legal}_{sid} = \langle \mathsf{Var}_{sid} \rangle_{\mathbb{Z}_p[\mathsf{Var}_{sid}]}$ corresponding to unobservable polynomials can be written as an ideal).

Efficiency Similar to $\mathcal{G}$-GG, the observable $\mathcal{G}$-oGG is also not efficient. In addition to sampling the random encoding functions τ_i at the start, the sets $\mathsf{R}_i[g]$ in $\mathcal{G}$-oGG can also blow up to superpoly sizes in the worst case. However, as we argue in Lemma 3, with overwhelming probability, $\mathsf{R}_i[g]$ will be a singleton. To make $\mathcal{G}$-oGG efficient, one can sample τ_i values lazily (as sketched in Functionality 17 in Appendix F), and if any set $\mathsf{R}_i[g]$ ever gets larger than a single element (which happens only with negligible probability), one can switch to an arbitrary error mode (e.g., stop maintaining R and instead make everything observable).

4 Switching to symbolic groups

The restricted global observable generic group functionality $\mathcal{G}$-oGG faithfully models a generic group (as in $\mathcal{G}$-GG) with tacked-on observation capabilities. However, an issue of $\mathcal{G}$-oGG when doing security proofs is that the session separation in $\mathcal{G}$-oGG is imperfect. It *can* happen that some group element belongs to two sessions in $\mathcal{G}$-oGG, and both sessions will be able to observe operations involving it. This is not desirable and will be an error event for most applications. In this section, we present the *symbolic* (restricted observable) generic group model $\mathcal{G}$-oSG, where session separation is *perfect* by definition and this error event cannot happen. Lemma 3 shows that $\mathcal{G}$-oGG can be securely replaced by $\mathcal{G}$-oSG.

In addition to that, $\mathcal{G}$-oSG will also support typical security proof techniques. Many typical (game-based) generic group model security proofs follow roughly (at least in spirit) this template:

1. Run the generic adversary, while the reduction answers its generic group oracle queries.
2. Argue that instead of sampling random discrete logarithm secrets $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_p^*$, the reduction can play the role of the generic group oracle using formal variables $\mathsf{X}_\alpha, \mathsf{X}_\beta$. Applying Schwartz–Zippel shows that this is undetectable to the generic adversary.
3. Argue that the adversary only makes linear (or pairing) operations, so whenever the adversary outputs a group element h^* corresponding to $a \cdot \mathsf{X}_\alpha + b\mathsf{X}_\beta$, the reduction algorithm can extract the discrete logarithm representation $(a, b) \in \mathbb{Z}_p^2$ of that group element by looking at the generic group oracle queries the adversary made.
4. Argue that the group elements output by the adversary do not threaten security because they are only linear combinations of the (polynomials corresponding to the) elements the reduction has provided (e.g., the adversary cannot output X if we only give it $\mathsf{X} + \mathsf{Y}$, but not Y).

The last step is highly dependent on the concrete scheme to be proven secure. For example, it can take the form of "We only give $\mathcal{A}$ the public key $[\mathsf{X}_x, \mathsf{X}_y]_2$ and signatures $\sigma_i = [\mathsf{X}_{r_i}, \mathsf{X}_{r_i}(\mathsf{X}_x + m_i\mathsf{X}_y)]_1$, so when the adversary outputs a forgery (in the first group), it must be of the form $[\sum a_i \cdot \mathsf{X}_{r_i} + \sum b_i \cdot \mathsf{X}_{r_i}(\mathsf{X}_x + m_i \mathsf{X}_y)]_1$, and hence cannot be forgery" [PS16]. These arguments are inherently *symbolic*, i.e. in the last step, $\mathsf{X}_x, \mathsf{X}_y, \mathsf{X}_{r_i}$ are formal variables, and the verification equation is an equation over polynomials in those variables. There are no concrete values anymore, and hence we are discussing the values and equations symbolically. In particular, this guarantees that there cannot be any accidental guesses of secret keys or randomness, meaning that proofs at this stage are usually *perfect*.

In this section, we extend $\mathcal{G}$-oSG in Functionality 5 to enable the proof strategy above as follows (the steps here correspond to the steps above).

1. Run the UC environment/adversary with $\mathcal{G}$-oGG replaced by $\mathcal{G}$-oSG.
2. Instead of choosing random secrets $\alpha, \beta \overset{\$}{\leftarrow} \mathbb{Z}_p^*$, have the UC simulator ask (the extended) $\mathcal{G}$-oSG for corresponding formal variables $\mathsf{X}_\alpha, \mathsf{X}_\beta \leftarrow \textsc{GetRnd}()$, and use $\textsc{ComputeSymbolic}_{sid}$ to output group elements relative to the secrets. Lemma 4 shows that this switch is undetectable.
3. Have the UC simulator use the algorithm FindRep to extract the discrete logarithm representation (a, b) from element h^*. In contrast to the typical generic group proofs, the UC simulator does not see all generic group operations, but Lemma 5 shows that the restricted observations are enough to get meaningful guarantees.
4. Argue that the group elements output by the adversary do not threaten security. This part is essentially the same as in standard generic group proofs. It is supported by $\mathcal{G}$-oSG (+ extensions), which automatically keeps track of the polynomial $\tau_i^{-1}(h)$ corresponding to each group element h.

The symbolic $\mathcal{G}$-oSG with extensions (Functionality 5) will allow most security proofs to conveniently hop to a setting where secrets are formal variables, group elements correspond one-to-one to polynomials (enabling *symbolic* analysis of group elements/operations), and the simulator can extract discrete logarithm representations. Most proofs can simply invoke our Lemmas 3 to 5 without ever applying Schwartz–Zippel themselves. We use this framework when proving Groth16 secure in Section 5.

4.1 The restricted observable global symbolic group model with perfect session separation

We introduce the restricted observable global symbolic group model $\mathcal{G}$-oSG in Functionality 4, which, in contrast to $\mathcal{G}$-oGG, has perfect separation of sessions. This separation is modeled similarly to $\mathcal{G}$-oGG, with polynomials. In contrast to $\mathcal{G}$-oGG, the polynomials will not only be some bookkeeping artifacts R alongside the actual $\mathcal{G}$-GG functionality, but rather the main driver behind group operations. More concretely, the random encoding function $\tau_i : \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}] \to S_i$ [12] now injectively maps *polynomials* f to random encodings h, rather than concrete discrete logarithms. In particular, this means that any group element (encoding) $h \in S_i$ has a unique polynomial $\tau_i^{-1}(h)$ associated with it, which also directly determines its behavior w.r.t. Op, Pair. In this sense, the polynomial mapping τ_i serves two purposes now: It manages the algebraic properties of group elements (managed by τ_i over $\mathbb{Z}_p$ in $\mathcal{G}$-oGG) *and* it is used to decide observability (used to be managed by R in $\mathcal{G}$-oGG).

A consequence of having τ_i map *polynomials* to S_i is that there exist no injective $\tau_i : \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}] \to S_i$. We cannot choose τ_i randomly at the beginning, anymore. For this reason, images of τ_i are lazily sampled via Tau. Because the adversary is computationally bounded, we will not run out of fresh unused images in $S_i \setminus \mathrm{im}(\tau_i)$ to use in Line 23.

[12] For now, ignore the list SimVar of formal variables. It is empty and will only be used in the $\mathcal{G}$-oSG extensions (Functionality 5).

Functionality 4: $\mathcal{G}$-oSG

Differences with $\mathcal{G}$-oGG are highlighted in purple. Values relevant only in the $\mathcal{G}$-oSG extensions (Functionality 5) are highlighted in yellow (can be ignored on first read).

- τ_i now maps polynomials ($\mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$ instead of $\mathbb{Z}_p$) to random encodings S_i
- $\mathsf{Var}_{i,sid}$ initially empty lists of polynomial variables
- SimVar_{sid} empty lists of polynomial variables $\mathsf{X}_{\mathrm{rnd}}$. Only used in Functionality 5
- SimVal_{sid} empty lists of random scalars $x_{\mathrm{rnd}} \in \mathbb{Z}_p$ corresponding to SimVar_{sid}
- Ob initially empty list of (globally) observable actions
- Ob_{sid} initially empty lists of all actions observable in specific session sid, including actions of parties *in* session sid (only read in the $\mathcal{G}$-oSG extensions)
- C_i initially empty sets $C_i \subseteq S_i$ of group elements that can be the basis for extraction (only read in the $\mathcal{G}$-oSG extensions)

Furthermore, we use the following terms derived from the current state

- We write Var_{sid}, Var as before. Similarly, SimVar is the concatenation of all SimVar_{sid}. Var_{-sid} is the concatenation of all $\mathsf{Var}_{sid'}$, where $sid' \neq sid$.
- $\mathsf{Legal}_{sid} = \langle \mathsf{Var}_{sid} \rangle_{\mathbb{Z}_p[\mathsf{Var}_{sid}, \mathsf{SimVar}_{sid}^{\pm 1}]} = \sum_{\mathsf{X} \in \mathsf{Var}_{sid}} \mathsf{X} \cdot \mathbb{Z}_p[\mathsf{Var}_{sid}, \mathsf{SimVar}_{sid}^{\pm 1}]$. // Legal_{sid} is the set of (Laurent) polynomials that contain only variables from Var_{sid} and SimVar_{sid} (with potentially negative exponents), where every nonzero term has some factor $\mathsf{X} \in \mathsf{Var}_{sid}$.

```
INIT()  // Invoked only upon creation
for $i \in \{1, 2, \mathrm{t}\}$ do
    $\tau_i \leftarrow \{\}$
    $\textsc{Tau}(i, 1)$
    $C_i \leftarrow C_i \cup \{1\}$

CANONICALGEN_{sid}(i)
return $\textsc{Tau}(i, 1)$

OBSERVE_{sid}()
return $\mathit{Ob}$

OP_{sid}(i, g_1, g_2, a_1, a_2)
assert $(g_1, g_2, a_1, a_2) \in S_i^2 \times \mathbb{Z}_p^2$
for $j \in \{1, 2\}$ do
    $\textsc{Touch}_{sid}(i, g_j)$
$\mathsf{f} \leftarrow a_1 \tau_i^{-1}(g_1) + a_2 \tau_i^{-1}(g_2)$
$h \leftarrow \textsc{Tau}(i, \mathsf{f})$
if $\mathsf{f} \notin \mathsf{Legal}_{sid}$ then
    $\mathit{Ob} \leftarrow \mathit{Ob} : [(\textsc{Op}, i, g_1, g_2, a_1, a_2, h)]$
    $\mathit{Ob}_{sid'} \leftarrow \mathit{Ob}_{sid'} : [(\textsc{Op}, i, g_1, g_2, a_1, a_2, h)]$ for all $sid'$ (incl. $sid$)
$\mathit{Ob}_{sid} \leftarrow \mathit{Ob}_{sid} : [(\textsc{Op}, i, g_1, g_2, a_1, a_2, h)]$
return $h$
```

```
TOUCH_{sid}(i, g)
if $g \notin \mathrm{im}(\tau_i)$ then
    Initialize a fresh variable $\mathsf{X}$
    $\mathsf{Var}_{i,sid} \leftarrow \mathsf{Var}_{i,sid} : [\mathsf{X}]$
    $\tau_i(\mathsf{X}) \leftarrow g$
    $C_i \leftarrow C_i \cup \{g\}$

TAU(i, f)  // internal
if $\tau_i(\mathsf{f}) = \perp$ then
    $\tau_i(\mathsf{f}) \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau_i)$
return $\tau_i(\mathsf{f})$

PAIR_{sid}(g_1, g_2)
assert $(g_1, g_2) \in S_1 \times S_2$
for $i \in \{1, 2\}$ do
    $\textsc{Touch}_{sid}(i, g_i)$
$\mathsf{f} \leftarrow \tau_1^{-1}(g_1) \cdot \tau_2^{-1}(g_2)$
$h \leftarrow \textsc{Tau}(\mathrm{t}, \mathsf{f})$
if $\mathsf{f} \notin \mathsf{Legal}_{sid}$ then
    $\mathit{Ob} \leftarrow \mathit{Ob} : [(\textsc{Pair}, \mathrm{t}, g_1, g_2, h)]$
    $\mathit{Ob}_{sid'} \leftarrow \mathit{Ob}_{sid'} : [(\textsc{Pair}, \mathrm{t}, g_1, g_2, h)]$ for all $sid'$ (incl. $sid$)
$\mathit{Ob}_{sid} \leftarrow \mathit{Ob}_{sid} : [(\textsc{Pair}, \mathrm{t}, g_1, g_2, h)]$
return $h$
```

The following lemma establishes that we can replace the procedures of $\mathcal{G}$-oGG with their idealized versions from $\mathcal{G}$-oSG (ignoring the "extra" procedures that $\mathcal{G}$-oSG carries).

Lemma 3. *Let* $\mathcal{O}_{\text{real}} = \mathcal{G}\text{-oGG}.[\textsc{CanonicalGen}, \textsc{Observe}, \textsc{Touch}, \textsc{Op}, \textsc{Pair}]$. *Let* $\mathcal{O}_{\text{symb}} = \mathcal{G}\text{-oSG}.[\textsc{CanonicalGen}, \textsc{Observe}, \textsc{Touch}, \textsc{Op}, \textsc{Pair}]$.

For all algorithms $\mathcal{B}$ *that make at most* q *oracle queries, it holds that*

$$\left|\Pr\left[\mathcal{B}^{\mathcal{O}_{\text{real}}} = 1\right] - \Pr\left[\mathcal{B}^{\mathcal{O}_{\text{symb}}} = 1\right]\right| \leq \binom{3q+1}{2} \cdot 2/p \leq (9q^2 + 3q)/p$$

When treating interfaces $\textsc{Interface}$ as oracles, this means that the caller specifies session sid and input x, then gets the result of $\textsc{Interface}_{sid}(x)$. The oracles share state.

Proof (Sketch). Both $\mathcal{G}$-oGG and $\mathcal{G}$-oSG use polynomial variables X to separate sessions and use polynomials (via τ_i^{-1} for $\mathcal{G}$-oSG and via R_i for $\mathcal{G}$-oGG) to make decisions about observability. The essential difference between $\mathcal{G}$-oGG and $\mathcal{G}$-oSG is that $\mathcal{G}$-oSG (1) keeps track of group elements in terms of variables X via $\tau_i : \mathbb{Z}_p[\mathsf{Var}] \to S_i$ and (2) makes decisions on whether to output a fresh random encoding or an old one w.r.t. the polynomials in τ_i. $\mathcal{G}$-oGG, in contrast, (1) keeps track of group elements in terms of random session-separating dlogs x via $\tau_i : \mathbb{Z}_p \to S_i$ and (2) makes decisions on whether to output a fresh random encoding or an old one w.r.t. the scalars in τ_i. The proof establishes that if there are no collisions when replacing the polynomial variables X in $\mathcal{G}$-oSG with random scalars x, then $\mathcal{G}$-oSG behaves exactly like $\mathcal{G}$-oGG. Schwartz-Zippel (Lemma 1) implies that collisions are rare because the $\leq 3(q+1)$ involved polynomials are of degree at most ≤ 2 and p is large. This description omits some subtleties in proving Lemma 3. For example, the scalars x in $\mathcal{G}$-oGG are not actually uniformly independently random, as required by Schwartz–Zippel, but rather uniform among yet-unused discrete logarithms (a set which stochastically depends on the random choice of other x). The full proof can be found in Appendix F.

Overall, as the first step in any $\mathcal{G}$-oGG proof, we expect $\mathcal{G}$-oGG to be replaced by $\mathcal{G}$-oSG, which is more convenient to handle in security proofs, and will enable powerful symbolic analysis using its extensions.

4.2 Extending $\mathcal{G}$-oSG with support for symbolic analysis

As sketched at the beginning of Section 4, our goal is to support typical GGM proof techniques in the $\mathcal{G}$-oSG UC setting. For this, we extend $\mathcal{G}$-oSG with additional interfaces in Functionality 5.

We first direct our attention at Functionality 5's interfaces $\textsc{GetRnd}$, $\textsc{ComputeConcrete}$, $\textsc{ComputeAtomic}$, and $\textsc{ComputeSymbolic}$. They model interaction of an algorithm $\mathcal{B}$ (usually the UC simulator) with hidden variables. They will allow us to make statements about changes in $\mathcal{B}$'s behavior as long as $\mathcal{B}$ does not use those hidden variables other than for group operations. The interfaces are to be used as follows: Whenever $\mathcal{B}$ generates a secret $\alpha \leftarrow \mathbb{Z}_p^*$, this can be modeled as a call to $\textsc{GetRnd}$, which samples α for $\mathcal{B}$, and returns a handle (in the form of a formal variable) X_α. $\mathcal{G}$-oSG keeps a list of these variables X_α in SimVar and the corresponding values (hidden from $\mathcal{B}$) in SimVal. In the following, $\mathcal{B}$ will use the handle X_α to describe computations involving α using Laurent polynomials $\mathsf{f}_j \in \mathbb{Z}_p[\mathsf{SimVar}^{\pm 1}]$. Whenever $\mathcal{B}$ would use α to compute some group element g, we can model this as a call to $\textsc{ComputeConcrete}$. It passes the description of the sum it wants to compute in the form of pairs $(h_j, \mathsf{f}_j) \in S_i \times \mathbb{Z}_p[\mathsf{SimVar}^{\pm 1}]$ as input to $\textsc{ComputeConcrete}$, which then uses its knowledge of the concrete values SimVal to compute "$h = \sum h_j \cdot \mathsf{f}_j(\mathit{Val})$" using the $\textsc{Op}$ oracle.

Functionality 5: $\mathcal{G}$-oSG extensions

This box contains interfaces in addition to the ones shown in Functionality 4. These interfaces are artifacts for security proofs rather than publicly available interfaces. They model interaction with unknown random values/-variables and the discrete logarithm representation extraction process via FindRep. See Lemmas 3 to 5 for how these interfaces are used.

```
$\underline{\textsc{GetRnd}_{sid}()}$
Initialize a new variable $\mathsf{X}$
$\mathsf{SimVar}_{sid} \leftarrow \mathsf{SimVar}_{sid} : [\mathsf{X}]$
$x \xleftarrow{\$} \mathbb{Z}_p^*$
$\mathit{SimVal}_{sid} \leftarrow \mathit{SimVal}_{sid} : [x]$
return $\mathsf{X}$

$\underline{\textsc{ComputeSymbolic}_{sid}(i, (h_j, \mathsf{f}_j)_{j=1}^n)}$
assert $\tau_i^{-1}(h_j) \in \mathsf{Legal}_{sid}$ // $h_j$ belongs to session $sid$ and $\mathsf{f}_j \in \mathbb{Z}_p[\mathsf{SimVar}_{sid}^{\pm 1}]$ for all $j \in [n]$.
$\mathsf{f} \leftarrow \sum_j \tau_i^{-1}(h_j) \cdot \mathsf{f}_j$ // $\in \mathbb{Z}_p[\mathsf{Var}_{sid}, \mathsf{SimVar}_{sid}^{\pm 1}]$
$h \leftarrow \textsc{Tau}(i, \mathsf{f})$
$C_i \leftarrow C_i \cup \{h\}$
return $h$

$\underline{\textsc{ComputeAtomic}_{sid}(i, (h_j, \mathsf{f}_j)_{j=1}^n)}$
assert $\tau_i^{-1}(h_j) \in \mathsf{Legal}_{sid}$ // $h_j$ belongs to session $sid$ and $\mathsf{f}_j \in \mathbb{Z}_p[\mathsf{SimVar}_{sid}^{\pm 1}]$ for all $j \in [n]$.
$\mathsf{f} \leftarrow \sum_j \tau_i^{-1}(h_j) \cdot \mathsf{f}_j(\mathit{SimVal}_{sid})$ // $\in \mathbb{Z}_p[\mathsf{Var}_{sid}]$
$h \leftarrow \textsc{Tau}(i, \mathsf{f})$
return $h$

$\underline{\textsc{ComputeConcrete}_{sid}(i, (h_j, \mathsf{f}_j)_{j=1}^n)}$
assert $\tau_i^{-1}(h_j) \in \mathsf{Legal}_{sid}$ // $h_j$ belongs to session $sid$ and $\mathsf{f}_j \in \mathbb{Z}_p[\mathsf{SimVar}_{sid}^{\pm 1}]$ for all $j \in [n]$.
$h \leftarrow \textsc{Tau}(i, 0)$ // $h = 0$ neutral element
for $j \in [n]$ do
    $a_j \leftarrow \mathsf{f}_j(\mathit{SimVal}_{sid})$ // $\in \mathbb{Z}_p$. Compute exponent $a_j$ from secrets $\mathit{SimVal}_{sid}$
    $h \leftarrow \textsc{Op}_{sid}(i, h, h_j, 1, a_j)$ // $h \leftarrow h + a_j \cdot h_j$
return $h$

$\underline{\textsc{GetRep}_{sid}(i, h^*, B)}$
assert $i \in \{1, 2\}, h^* \in \mathrm{im}(\tau_i), B \in (C_i)^n$ with $B_j \neq B_\ell$ for $j \neq \ell$.
$(a_j)_{j=1}^n \leftarrow \textsc{FindRep}(i, h^*, \mathit{Ob}_{sid}, B)$
$\mathsf{V} = \sum_{j=1}^n a_j \cdot \tau_i^{-1}(B_j)$ // Result as polynomial $\mathsf{V} \in \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$
assert $\exists \mathsf{b}_j, \mathsf{c}_j \in \mathbb{Z}_p[\mathsf{SimVar}^{\pm 1}]$ : $\mathsf{V} = \tau_i^{-1}(h^*) + \mathsf{foreign} + \mathsf{missing}$, where $\mathsf{foreign} = \sum_{\mathsf{X}_j \in \mathsf{Var}_{-sid}} \mathsf{b}_j \mathsf{X}_j$ and $\mathsf{missing} = \sum_{j : C_i[j] \notin B} \mathsf{c}_j \cdot \tau_i^{-1}(C_i[j])$, where $C_i[j]$ is the $j$th element of the set $C_i$ according to some canonical ordering.
return $a_1, \ldots, a_n$
```

ComputeConcrete is indistinguishable from ComputeSymbolic. In the latter, the computation is done both *atomically* in a single step, and, more importantly, *symbolically*, meaning that ComputeSymbolic does not access the concrete values *SimVal* at all. Instead, it simply computes the result f in terms of polynomials, and then returns $\textsc{Tau}(i, \mathsf{f})$. This functionality heavily uses the fact that the encoding functions τ_i already work over polynomials. In the original $\mathcal{G}$-oSG, this capability is only used for the sake of domain separation (with the Var variables), but in the presence of ComputeSymbolic, it is also used to make computations directly over formal variables X_α corresponding to secrets of $\mathcal{B}$. For example, if g is a generator corresponding to $\mathsf{X}_g \in \mathsf{Var}$, and the computation is "$h \leftarrow \alpha^{-2} \cdot g$", then the result h will be internally associated with the polynomial $\mathsf{f} = \mathsf{X}_\alpha^{-2} \cdot \mathsf{X}_g = \tau_i^{-1}(h) \in \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$, and it will algebraically behave like f.

As an intermediate step between interfaces ComputeConcrete and ComputeSymbolic, the interface ComputeAtomic does the computation in ComputeConcrete, but using only a single query to Tau.

Overall, this enables the security proof to talk about group elements h by their polynomial representation $\tau_i^{-1}(h)$, which is a powerful analysis tool. The following lemma establishes indistinguishability between the three computation methods.

Lemma 4. *Let* $\mathcal{O} = \mathcal{G}\text{-oSG}.[\textsc{CanonicalGen}, \textsc{Observe}, \textsc{Touch}, \textsc{Op}, \textsc{Pair}, \underline{\textsc{GetRnd}}]$. *Let* $\mathcal{B}^{\mathcal{O}, \textsc{ComputeX}}$ *be an algorithm that makes at most q oracle queries. For oracle queries*

$$\textsc{ComputeX}(i, (h_{\ell,j}, \mathsf{f}_{\ell,j})_{j=1}^{n_\ell}),$$

let $q' \geq \sum_{\ell=1}^q n_\ell$ *be (an upper bound for) the number of supplied polynomials to the last oracle. Let* $d \geq \max_{i,h}(\deg(\tau_i^{-1}(h)))$ *be (an upper bound for) the maximum degree of (Laurent)* *polynomials in the execution of* $\mathcal{B}^{\mathcal{O},\textsc{ComputeSymbolic}}$ *If* $3q + q' + 1 \leq p$*, then*

$$\left| \begin{array}{l} \Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeConcrete}} = 1\right] \\ -\Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeAtomic}} = 1\right] \end{array} \right| \leq (2q + q') \cdot q'/(p - q)$$

$$\left| \begin{array}{l} \Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeAtomic}} = 1\right] \\ -\Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeSymbolic}} = 1\right] \end{array} \right| \leq \binom{3q+1}{2} \cdot 2d/(p-1)$$

As a consequence of the lemma, we get this bound for applicable $\mathcal{B}$:

$$\left|\Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeConcrete}} = 1\right] - \Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeSymbolic}} = 1\right]\right| \quad \leq (2q+q')\cdot\frac{3q'}{2p}+(9q^2+3q)d/(p-1).$$

Proof. For the first part of the proof, replacing ComputeConcrete with ComputeAtomic cannot be detected by $\mathcal{B}$ unless it successfully guesses an intermediate result's random encoding and queries it to Touch or ComputeConcrete / ComputeAtomic. The chances for guessing one of the less than q' intermediate results among all possible p, of which at most q can be ruled out a priori because they have been output of some other query, are at most $q'/(p - q)$. $\mathcal{B}$ makes at most $2q + q'$ guesses, giving us the bound in the lemma. See Appendix G for the full proof.

For the second part, replacing the interface ComputeAtomic with ComputeSymbolic cannot be detected unless there is a collision among Laurent polynomials with random input *SimVal*, i.e. two polynomials $\mathsf{f} \neq \mathsf{f}' \in \mathrm{dom}(\tau_i) \subset \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$ such that $\mathsf{f}(Val) = \mathsf{f}'(Val) \in \mathbb{Z}_p[\mathsf{Var}]$. Note that we are not interested in whether the session-separation variables Var collide — those remain symbolic in both settings. This is a straightforward application of Lemma 2. Consider any two polynomials $\mathsf{f} \neq \mathsf{f}' \in \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$ queried to $\textsc{Tau}(i, \cdot)$ for $i \in \{1, 2\}$. By virtue of generic group operations, we can write $\mathsf{f} = \sum_j \mathsf{X}_j \cdot \mathsf{t}_j + \mathsf{t}_0$ and $\mathsf{f}' = \sum_j \mathsf{X}_j \cdot \mathsf{t}'_j + \mathsf{t}'_0$, where $\mathsf{X}_j \in \mathsf{Var}$ and $\mathsf{t}_j, \mathsf{t}'_j \in \mathbb{Z}_p[\mathsf{SimVar}^{\pm 1}]$. Because $\mathsf{f} \neq \mathsf{f}'$, there must be some $\mathsf{t}_j \neq \mathsf{t}'_j$. From Lemma 2, we know that $\Pr[\mathsf{t}_j(SimVal) = \mathsf{t}'_j(SimVal)] \leq 2d/(p-1)$. Hence $\Pr[\mathsf{f}(SimVal) = \mathsf{f}'(SimVal)] \leq 2d/(p-1)$. For polynomials $\mathsf{f} = \sum_{j,\ell} \mathsf{X}_j\mathsf{X}_\ell \cdot \mathsf{t}_{j,\ell} + \sum_j \mathsf{X}_j \cdot \mathsf{t}_{j,0} + \mathsf{t}_0$ belonging to the target group, the same argument holds, i.e. $\mathsf{f} \neq \mathsf{f}' \Rightarrow \Pr[\mathsf{f}(SimVal) = \mathsf{f}'(SimVal)] \leq 2d/(p - 1)$.

If no such collision happens, then the ComputeSymbolic setting behaves exactly like the ComputeAtomic setting. There are at most $\binom{3q+1}{2}$ pairs $\mathsf{f} \neq \mathsf{f}'$ of polynomials, so by the union bound, $\Pr[\exists i, \{\mathsf{f}, \mathsf{f}'\} \in \binom{\mathrm{dom}(\tau_i)}{2} : \ \mathsf{f}(Val)] \leq \binom{3q+1}{2} \cdot 2d/(p - 1)$. □

4.3 Extracting discrete logarithm representations

Finally, in generic group model proofs, one usually wants to extract the discrete logarithm representations of certain group elements. In the UC setting with a global generic group, this is complicated by the fact that the UC simulator for session *sid* does not have access to *all* GGM queries, but only to "illegal" queries made in foreign sessions $sid' \neq sid$ (Line 12 and 24 in Functionality 3), and to queries made by the adversary in session *sid* (by design of UC / the default identity bound ξ). The list of observations available to the simulator is modeled in Line 15, 14 and Line 33 and 32 of Functionality 4. Some operations are, by design, unobservable. For example, if a protocol (embodied by the environment) in session sid' computes an element $\mathsf{f} = 3\mathsf{X} \in \mathsf{Legal}_{sid'}$, then the simulator in session *sid* does not get any information about that computation, and will consequently not be able to extract the coefficient 3.

The GetRep interface (Functionality 5), defines in Line 58 what we can expect from the algorithm FindRep given the limited observation information: When extracting a representation for h^*, the algorithm FindRep outputs coefficients that (together with the basis)

Function 1: FindRep

```
FindRep($i, h^*, Ob_{sid}, B$)
1: // Finds representation of $h^* \in S_i$ w.r.t. basis $B \in S_i^n$. Requires observations $Ob_{sid}$ of globally observable
   operations and the simulator's operations (see Functionality 5)
2: // Returns a (partial) representation $Rep[h^*] \in \mathbb{Z}_p^n$ in the form of coefficients for basis elements
3: assert $i \in \{1,2\}$ // FindRep for target group in Appendix E
4: Parse $B = (B_1, \dots, B_n) \in S_i^n$ // Basis elements for the representation
5: $Rep[h] \leftarrow 0^n \in \mathbb{Z}_p^n$ initially for all $h$
6: for $j \in [n]$ do $Rep[B_j] \leftarrow (\text{Kronecker}_{\ell,j})_{\ell=1}^n$ // $\in \mathbb{Z}_p^n$
7: for $ob = (\text{Op}, i, g_1, g_2, a_1, a_2, h) \in Ob_{sid}$ do // Observed operations in order of $Ob_{sid}$ (filtered by Op, $i$)
8:     $Rep[h] \leftarrow a_1 \cdot Rep[g_1] + a_2 \cdot Rep[g_2]$ // Update representation of $h$ w.r.t. to operation result "$h = a_1 g_1 + a_2 g_2$"
   return $Rep[h^*]$ // Return representation for the $h^*$ we were interested in
```

almost sum up to the polynomial $\tau_i^{-1}(h^*)$. What is *missing* from that sum can only be (1) foreign terms, that contain foreign variables X_j from another session (because those terms may be subject to unobservable computations), and (2) missing terms, which contain a variable X not supplied to FindRep as a basis (because FindRep has no starting point to find coefficients for X from). When doing security proofs, one would usually argue that those terms are not required for the simulator to successfully do its job. For example, the Groth16 simulator, when extracting a Groth16 proof, is only interested in (1) elements on the correct basis (proofs containing another basis are rejected by the verification equation), and (2) coefficients of one specific term of the proof's polynomial representation, which correspond to the witness.

The FindRep algorithm (Function 1) itself is quite simple: it linearly scans the list of observations and keeps track of their representations *Rep* in terms of the basis B supplied.

The following lemma states that FindRep works correctly. This is defined in terms of the symbolic computation setting and the interface GetRep, which runs FindRep with the expected input (in particular with the correct observation list Ob_{sid}) and then checks the output.

Lemma 5. *Consider* $\mathcal{O} = \mathcal{G}\text{-}\mathtt{oSG}.$[CanonicalGen, Observe, Touch, Op, Pair, GetRnd, <u>ComputeSymbolic</u>, <u>GetRep</u>]. *Let* $\mathcal{B}$ *be an algorithm that makes at most* p *queries. Then*

$$\Pr\left[\mathcal{B}^{\mathcal{O}} \textit{ has assertion in Line 58 of Functionality 5 fail}\right] = 0$$

The proof can be found in Appendix I.

5 UC security of Groth16

In Protocol 1, we present the Groth16 protocol $\Pi\text{-}\mathtt{G16}$ in the presence of our global observable generic group functionality $\mathcal{G}\text{-}\mathtt{oGG}$. The protocol is described in the $\mathcal{F}\text{-}\mathtt{CRS}$-hybrid model (Functionality 6). The crucial operation is for-loop starting at Line 1, in which $\mathcal{F}\text{-}\mathtt{CRS}$ registers uniformly random session-specific generators $g_{sid,i}$. In this way, all of the group operations performed by honest provers are confined to the domain of the current session and thus unobservable by the environment (except if $\mathcal{F}\text{-}\mathtt{CRS}$ or prover accidentally operates on group elements that are already reserved for another session, which occurs with negligible probability).

Theorem 1. $\Pi\text{-}\mathtt{G16}$ *UC-realizes* $\mathcal{F}\text{-}\mathtt{wNIZK}$ *in the* $\mathcal{F}\text{-}\mathtt{CRS}$*-hybrid model in the presence of* $\mathcal{G}\text{-}\mathtt{oGG}$. *Concretely, for any PPT adversary* $\mathcal{A}$*, there exists a PPT simulator* $\mathcal{S}_{\mathtt{G16}}$ *such that*

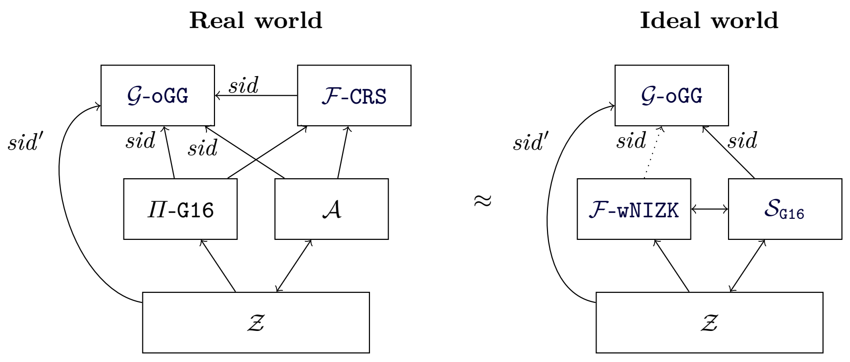

Fig. 1: An illustration of the real and ideal world settings for Theorem 1 and its proof. We omit the dummy parties for $\mathcal{F}$-wNIZK.

for every $\mathcal{Z}$ that makes at most $q_\mathcal{Z}$ queries to $\mathcal{G}$-oGG, $q_\mathcal{P}$ queries to the Prove *interface, and $q_\mathcal{V}$ queries to the* Verify *interface,*

$$\begin{aligned}&|\Pr[\mathsf{EXEC}_{\mathcal{F}\text{-wNIZK},\mathcal{Z},\mathcal{S}_{\texttt{G16}},\mathcal{G}\text{-oGG}}(\lambda, z) = 1] - \Pr[\mathsf{EXEC}_{\Pi\text{-G16},\mathcal{Z},\mathcal{A},\mathcal{G}\text{-oGG}}(\lambda, z) = 1]|\\ &\le 72 \cdot d \cdot (m + d + q_\mathcal{Z} + (m + d)q_\mathcal{P} + \ell q_\mathcal{V} + 1)^2/(p - 1)\end{aligned}$$

and $\mathcal{S}_{\texttt{G16}}$ performs in total the following operations:

- *at most $3q_\mathcal{P} + 9q_\mathcal{V} + 2q_\mathcal{Z} + 3d + m + 8$ queries to $\mathcal{G}$-oGG*
- *at most $(2\ell + 8)q_\mathcal{P} + (3q_\mathcal{Z} + 2\ell + 2)q_\mathcal{V} + (d + 1)(3m + 11)$ field operations where d, m, ℓ depend on the circuit size (see Functionality 6).*

Proof. We first construct a simulator $\mathcal{S}_{\texttt{G16}}$ described in Simulator 1. $\mathcal{S}_{\texttt{G16}}$ consists of two major components: Simulate that simulates proof (A, B, C) using $\mathcal{G}$-oGG.Op and a secret trapdoor for CRS, and Extract that extracts valid witness upon receiving a statement-proof pair using the Observe interface of $\mathcal{G}$-oGG and FindRep (Function 1). Whenever $\mathcal{Z}$ queries $\mathcal{G}$-oGG in the session with sid, $\mathcal{S}_{\texttt{G16}}$ forwards its queries to the corresponding wrapper interfaces, and relays back the responses to $\mathcal{Z}$. By simply counting the number of calls to $\mathcal{G}$-oGG interfaces and local addition, multiplicaiton, and division operations in $\mathbb{F}_p$ performed by $\mathcal{S}_{\texttt{G16}}$, we obtain the runtime of $\mathcal{S}_{\texttt{G16}}$ stated in the theorem (note that we provide the overall runtime of $\mathcal{S}_{\texttt{G16}}$ taking into account the number of activations through every interface: Init is called at most once, Simulate is called at most $q_\mathcal{P}$ times, Extract is called at most $q_\mathcal{V}$ times, and wrapper interfaces for $\mathcal{G}$-oGG are called at most $q_\mathcal{Z}$ times, respectively). We define a sequence of hybrids, starting from the ideal run of Groth16 with respect to $\mathcal{S}_{\texttt{G16}}$, $\mathcal{F}$-wNIZK in the presence of $\mathcal{G}$-oGG (see **Ideal world** of Fig. 1). The order of hybrids is relatively standard and a similar strategy appeared in the literature e.g. [Gro06].

- Hybrid H_0: This is equivalent to the ideal UC experiment with respect to $\mathcal{S}_{\texttt{G16}}$ and $\mathcal{F}$-wNIZK in the presence of $\mathcal{G}$-oGG. The distribution of the output of $\mathcal{Z}$ in H_0 is identical to $\mathsf{EXEC}_{\mathcal{F}\text{-wNIZK},\mathcal{Z},\mathcal{S}_{\texttt{G16}},\mathcal{G}\text{-oGG}}$.
- Hybrid H_1: Same as H_0 except that $\mathcal{G}$-oGG is replaced with its symbolic counterpart $\mathcal{G}$-oSG.
- Hybrid H_2: Same as H_1 except that $\mathcal{F}$-wNIZK is replaced with $\mathcal{F}$-wNIZK$'$, described in Functionality 20. The difference is that $\mathcal{F}$-wNIZK$'$ returns the output of the honest verification algorithm as in Π-G16 whenever its Verify interface gets invoked, while its Prove interface remains unchanged.
- Hybrid H_3: Same as H_2 except that $\mathcal{F}$-wNIZK$'$ is replaced with $\mathcal{F}$-wNIZK$''$, described in Functionality 21. The difference is that $\mathcal{F}$-wNIZK$''$ produces $\pi = (A, B, C)$ following the

Functionality 6: $\mathcal{F}$-CRS

$\mathcal{F}$-CRS has access to $\mathcal{G}$-oGG.
$\mathcal{F}$-CRS is parameterized by an NP-relation determined by QAP $(u_i, v_i, w_i)_{i=0}^m \in \mathbb{F}_p^{d-1}[X]$ and $t \in \mathbb{F}_p^d[X]$, where d is the number of multiplication gates and $a_0 = 1$:

$$\mathcal{R}_{\mathtt{QAP}} = \left\{(\{a_i\}_{i=1}^{\ell}, \{a_i\}_{i=\ell+1}^m) : (\textstyle\sum_{i=0}^m a_i u_i)(\sum_{i=0}^m a_i v_i) \equiv (\sum_{i=0}^m a_i w_i) \bmod t\right\}$$

To simplify notation we denote $q_i(\alpha, \beta, x) := \beta u_i(x) + \alpha v_i(x) + w_i(x)$.
$\mathcal{F}$-CRS stores state:

– σ, labels for common reference string

We use the following compact notation for a vector of encoded group elements with known discrete logs:

– $[x, y, \ldots]_{sid,i} := (\mathcal{G}\text{-oGG}.\textsc{Op}_{sid}(i, g_{sid,i}, g_{sid,i}, x, 0), \mathcal{G}\text{-oGG}.\textsc{Op}_{sid}(i, g_{sid,i}, g_{sid,i}, y, 0), \ldots)$

```
$\textsc{Init}_{sid}()$  // Invoked only upon creation
1: for $i = 1, 2$ do
2:     $g_{sid,i} \xleftarrow{\$} S_i$
3:     $\mathcal{G}$-oGG.$\textsc{Touch}_{sid}(i, g_{sid,i})$
4: $x, \alpha, \beta, \gamma, \delta \xleftarrow{\$} \mathbb{Z}_p$
5: $\sigma_1 \leftarrow [\alpha, \beta, \delta, \{x^i\}_{i=0}^{d-1}, \{q_i(\alpha, \beta, x)\gamma^{-1}\}_{i=0}^{\ell}, \{q_i(\alpha, \beta, x)\delta^{-1}\}_{i=\ell+1}^{m}, \{x^i t(x)\delta^{-1}\}_{i=0}^{d-2}]_{sid,1}$
6: $\sigma_2 \leftarrow [\beta, \gamma, \delta, \{x^i\}_{i=0}^{d-1}]_{sid,2}$
7: $\sigma \leftarrow (\sigma_1, \sigma_2)$

$\textsc{GetCRS}_{sid}()$
8: return $\sigma$
```

Protocol 1: Π-G16

The protocol has access to $\mathcal{F}$-CRS and $\mathcal{G}$-oGG.

```
$\textsc{Prove}_{sid}(x = \{a_i\}_{i=1}^{\ell}, w = \{a_i\}_{i=\ell+1}^{m})$
1: if $(x, w) \notin \mathcal{R}_{\mathtt{QAP}}$ then return $\perp$
2: $\sigma \leftarrow \mathcal{F}$-CRS$[\mathcal{G}$-oGG$, \mathcal{R}_{\mathtt{QAP}}]$.$\textsc{GetCRS}_{sid}()$
3: $r, s \xleftarrow{\$} \mathbb{Z}_p$
4: Compute $h \in \mathbb{F}^{d-2}[X]$ such that $ht = (\sum_{i=0}^m a_i u_i)(\sum_{i=0}^m a_i v_i) - (\sum_{i=0}^m a_i w_i)$
5: $A := [a]_{sid,1} \leftarrow \left[\sum_{i=0}^m a_i u_i(x) + \alpha + r\delta\right]_{sid,1}$ // Computed by calling $\mathcal{G}$-oGG.$\textsc{Op}_{sid}$ on $[x^i]_{sid,1}, [\alpha]_{sid,1}, [\delta]_{sid,1}$
6: $B := [b]_{sid,2} \leftarrow \left[\sum_{i=0}^m a_i v_i(x) + \beta + s\delta\right]_{sid,2}$ // Computed by calling $\mathcal{G}$-oGG.$\textsc{Op}_{sid}$ on $[x^i]_{sid,2}, [\beta]_{sid,2}, [\delta]_{sid,2}$
7: $C := [c]_{sid,1} \leftarrow \left[\sum_{i=\ell+1}^m a_i q_i(\alpha, \beta, x)\delta^{-1} + h(x)t(x)\delta^{-1} + sa + rb - rs\delta\right]_{sid,1}$ // Computed by calling $\mathcal{G}$-oGG.$\textsc{Op}_{sid}$ on $[q_i(\alpha, \beta, x)\delta^{-1}]_{sid,1}, [x^i t(x)\delta^{-1}]_{sid,1}, [a]_{sid,1}, [\beta]_{sid,1}, [\delta]_{sid,1}$
8: return $(A, B, C)$

$\textsc{Verify}_{sid}(x = \{a_i\}_{i=1}^{\ell}, \pi = (A, B, C))$
9: $\sigma \leftarrow \mathcal{F}$-CRS$[\mathcal{G}$-oGG$, \mathcal{R}_{\mathtt{QAP}}]$.$\textsc{GetCRS}_{sid}()$
10: $C_{\text{pub}} \leftarrow \left[\sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x)\gamma^{-1}\right]_{sid,1}$ // Computed by calling $\mathcal{G}$-oGG.$\textsc{Op}_{sid}$ on $[q_i(\alpha, \beta, x)\gamma^{-1}]_{sid,1}$
11: return $A \cdot B = C_{\text{pub}} \cdot [\gamma]_{sid,2} + C \cdot [\delta]_{sid,2} + [\alpha]_{sid,1} \cdot [\beta]_{sid,2}$ // Computed by calling $\mathcal{G}$-oGG.$\textsc{Op}_{sid}$ and $\mathcal{G}$-oGG.$\textsc{Pair}_{sid}$
```

honest prover algorithm as in Π-G16 whenever its PROVE interface gets invoked, instead of asking $\mathcal{S}_{\texttt{G16}}$ to simulate π.

– Hybrid H_4: Same as H_3 except that $\mathcal{G}$-oSG is replaced with its non-symbolic counterpart $\mathcal{G}$-oGG.

Note that H_4 is equivalent to the real execution of Π-G16 in $\mathcal{F}$-CRS-hybrid model in the presence of $\mathcal{G}$-oGG modulo minor syntactic differences. [13]

We defer the proof of the following supporting claims to Appendix J. We provide a sketch of each claim here:

– To prove $H_0 \approx H_1$ (Claim 4) and $H_3 \approx H_4$ (Claim 7) are indistinguishable, we can rely on Lemma 3 which generically bounds the loss incurred by replacing $\mathcal{G}$-oGG with $\mathcal{G}$-oSG. Now that H_1, H_2, H_3 only use $\mathcal{G}$-oSG, the discrete logs of session-specific generators $g_{sid,1}$ and $g_{sid,2}$ are treated as formal variables $\mathsf{X}_{sid,1}$ and $\mathsf{X}_{sid,2}$, respectively.

– To prove $H_1 \approx H_2$ (Claim 5), we first observe that $\mathcal{Z}$ distinguishes H_1 and H_2 only if the VERIFY interface receives accepting (x, π) such that x has never been queried to the PROVE interface. Thus, proving this exceptional event happens with negligible probability boils down to weak simulation-extractability of Groth16, which is already analyzed in [BKSV21]. To rely on the proof of [BKSV21] in a purely symbolic manner, we first switch to an intermediate hybrid in which $\mathcal{F}$-CRS aborts if it accidentally picks $g_{sid,1}$ and $g_{sid,2}$ that are already reserved for another session. As these elements are picked uniformly, this event occurs with negligible probability. Then we syntactically change the behavior of $\mathcal{S}_{\texttt{G16}}$ such that it treats randomness $\alpha, \beta, \ldots$ used for CRS generation and μ, ν for proof simulation as formal variables $\mathsf{X}_\alpha, \mathsf{X}_\beta, \ldots, \mathsf{X}_\mu, \mathsf{X}_\nu$, and then performs group operations using the COMPUTECONCRETE extension introduced in Section 4. In the next sub-hybrid, every invocation of COMPUTECONCRETE is replaced with COMPUTESYMBOLIC, enabled by Lemma 4. Once every randomness is fully treated as a formal variable, by Lemma 5, we have that the representation of $\pi = (A, B, C)$ output by the environment can be extracted without any error. Finally, we invoke the analysis of [BKSV21] to argue that extracted representation coincides with a valid witness. Towards this end, we additionally show that group elements from foreign sessions do not interfere with extraction of witnesses.

– To prove $H_2 \approx H_3$ (Claim 6), we mainly rely on the perfect ZK property of Groth16. However, a subtle issue arises in our G-GGM: a sequence of group operations performed by the simulator is different from that of the honest prover algorithm. Since these operations are also tracked by $\mathcal{G}$-oSG, there's a small chance that $\mathcal{Z}$ notices such inconsistent "styles" of group operations through the queries to $\mathcal{G}$-oSG. We show that this change is unnoticeable by invoking Lemma 7.

6 Composition when unobservability is required

The observable G-GGM is well suited for proving succinct arguments such as Groth16. In such schemes honest parties do not execute secret-dependent computations on adversarial group elements. As honest provers only compute on group elements originating from their own session, observability does not pose any privacy challenges, e.g. for the proof of the zero-knowledge property.

This situation is significantly different for other cryptographic schemes. For instance for the PAKE proof of [CNPR22] the authors assume that no information about oracle usage

[13] Concretely, to turn H_4 into $\mathsf{EXEC}_{\Pi\text{-G16},\mathcal{Z},\mathcal{A},\mathcal{G}\text{-oGG}}$ one can apply the following syntactic modifications: (1) CRS generation handled by $\mathcal{S}_{\texttt{G16}}$ is replaced with $\mathcal{F}$-CRS, (2) $\mathcal{F}$-wNIZK$''$ is viewed as Π-G16, and (3) $\mathcal{S}_{\texttt{G16}}$ is replaced with $\mathcal{A}$. Note that (3) is justified because SIMULATE and EXTRACT interfaces are not used at all in H_4, and the calls to the wrapper interfaces can be directly forwarded to $\mathcal{G}$-oGG.

is disclosed between parties. Similar issues arise for public-key encryption and oblivious PRFs [JKK14] when modeled with $\mathcal{G}$-`oGG`. The security proofs of such schemes fail when using $\mathcal{G}$-`oGG`, because the environment can send group elements—ciphertexts or blinded evaluation points—that originate from a foreign session. As an honest party applies their secret key to them, this leaks the key.

Note that this is inherent for any observable model of generic groups, as long as sessions are treated "symmetrically". That is, the $\textsc{Observe}_{sid}$ oracle can either be called by the simulator to prove session *sid* secure, or by the environment to model another protocol in session sid' composed in parallel, and prove overall security when reusing the same group.

Consider two cryptographic schemes: G16 in session *sid* and in session sid' a CCA2-secure variant of ElGamal, which we refer to as EG2, e.g. ECIES [Sma01] or Cramer-Shoup [CS98]. The distinguishing environment against G16 can make calls to $\textsc{Op}_{sid'}$. The $\textsc{Observe}_{sid}$ oracle must include $\textsc{Op}_{sid'}$ operations on group elements that originated in session *sid*, such as those used to generate a reference string for G16. Otherwise the extractor for G16 would fail to extract the witness. However, a distinguishing environment against EG2 (which can call $\textsc{Observe}_{sid}$) must *not* observe $\textsc{Op}_{sid'}$ operations on group elements that originated in session *sid*. Otherwise it would obtain leaked information about the EG2 secret key.

The crucial step to escape this conundrum is to observe that $\textsc{Observe}_{sid}$ is only called by the Groth16 simulator in the *ideal* world. Thus conceptually, we can work with a non-observable generic group (and apply the standard UCGS composition theorem to protocols like Π-`EG2` in that setting). Only when we want to switch from the concrete protocol Π-`G16` to the ideal $\mathcal{F}$-`wNIZK`, we switch to observable groups (as required by the Groth16 ideal world simulator). This is depicted in Fig. 2 (with details being developed in the following).

For this idea to work, we need a notion of *evolving* a global subroutine (like $\mathcal{G}$-`oGG`) over time, so that we can have an unobservable version of $\mathcal{G}$-`oGG` when it comes to applying the composition theorem to Π-`EG2` and an observable version when it comes to Π-`G16`. To model the observable/unobservable versions of $\mathcal{G}$-`oGG`, we introduce $\mathcal{G}$-`oGG`$[\mathfrak{S}]$ (Functionality 7), parameterized with a set of sessions $\mathfrak{S}$. This new functionality $\mathcal{G}$-`oGG`$[\mathfrak{S}]$ works like $\mathcal{G}$-`oGG` except that it allows only callers from sessions $sid \in \mathfrak{S}$ to see the observation list. In particular, $\mathcal{G}$-`oGG`$[\varnothing]$ behaves like the strict (unobservable) $\mathcal{G}$-`GG`.

However, note that unfortunately, we cannot weaken observability by simply replacing $\mathcal{G}$-`oGG`$[\mathfrak{S}]$ with $\mathcal{G}$-`oGG`$[\mathfrak{S}\setminus\mathfrak{S}^-]$. This is because the environment can easily distinguish $\mathcal{G}$-`oGG`$[\mathfrak{S}]$ from $\mathcal{G}$-`oGG`$[\mathfrak{S} \setminus \mathfrak{S}^-]$ by trying to query $\textsc{Observe}_{sid}$ using some $sid \in \mathfrak{S}^-$. This query would succeed in the first case, but not in the second. To solve this, we employ the identity bound ξ to disallow the environment from querying $\textsc{Observe}_{sid}$ on any session $sid \in \mathfrak{S}^-$. We get the following lemma, stating that with the identity bound, one can remove sessions unnoticed.

Lemma 6. *Let $\mathfrak{S}^- \subseteq \mathfrak{S}$. Let $\mathcal{Z}$ be an algorithm that does not query $\textsc{Observe}_{sid}$ for $sid \in \mathfrak{S}^-$. Then $\mathcal{Z}^{\mathcal{G}\text{-}\mathtt{oGG}[\mathfrak{S}]} \approx \mathcal{Z}^{\mathcal{G}\text{-}\mathtt{oGG}[\mathfrak{S}\setminus\mathfrak{S}^-]}$.*

The proof of this lemma is trivial. Note that to switch off observability completely, one can choose $\mathfrak{S}^- = \mathfrak{S}$. To switch off observability partially (e.g., to apply composition to additional schemes that require observations), one would choose a smaller $\mathfrak{S}^-$ (e.g., to leave sessions of additional schemes in $\mathfrak{S} \setminus \mathfrak{S}^-$).

Additionally, in order to make sure the Groth16 simulator can call $\textsc{Observe}_{sid}$ on the evolved $\mathcal{G}$-`oGG`$[\mathfrak{S}]$, we need to ensure that the session of any instance of Π-`G16` (and, consequently, $\mathcal{F}$-`wNIZK`) is one of the allowed sessions $sid \in \mathfrak{S}$. For this, we simply restrict Π-`G16` and $\mathcal{F}$-`wNIZK` to work only when instantiated with sessions $sid \in \mathfrak{S}$. We thus consider vari-

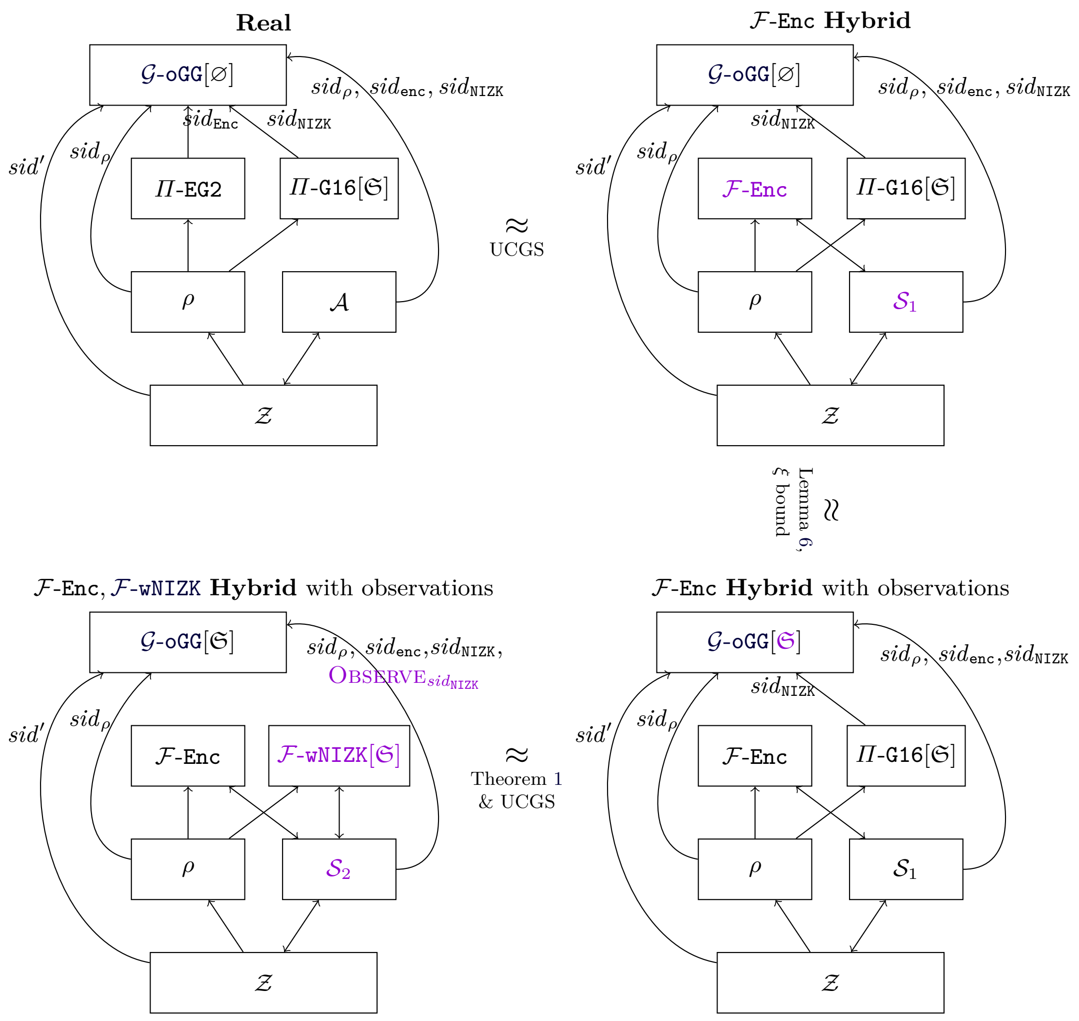
Real
$\mathcal{G}$-oGG[$\varnothing$]
sid_ρ, $sid_{\mathtt{enc}}$, $sid_{\mathtt{NIZK}}$
$sid_{\mathtt{Enc}}$
$sid_{\mathtt{NIZK}}$
sid'
sid_ρ
Π-EG2
Π-G16[$\mathfrak{S}$]
ρ
$\mathcal{A}$
$\mathcal{Z}$
$\approx$ UCGS
$\mathcal{F}$-Enc Hybrid
$\mathcal{F}$-Enc
$\mathcal{S}_1$
$\approx$ Lemma 6, ξ bound
$\mathcal{F}$-Enc, $\mathcal{F}$-wNIZK Hybrid with observations
$\mathcal{G}$-oGG[$\mathfrak{S}$]
sid_ρ, $sid_{\mathtt{enc}}$, $sid_{\mathtt{NIZK}}$, OBSERVE$_{sid_{\mathtt{NIZK}}}$
$\mathcal{F}$-wNIZK[$\mathfrak{S}$]
$\mathcal{S}_2$
$\approx$ Theorem 1 & UCGS
$\mathcal{F}$-Enc Hybrid with observations

Fig. 2: An illustration of composition, to be read starting top left, clockwise. Changes are highlighted in color. $\mathcal{F}$-CRS and dummy parties are omitted for simplicity.

Functionality 7: $\mathcal{G}$-oGG[$\mathfrak{S}$]

Parameterized with set $\mathfrak{S}$ of sessions that are allowed to call OBSERVE.

```
INIT, CANONICALGEN$_{sid}$, OP$_{sid}$, PAIR$_{sid}$, TOUCH$_{sid}$
as in $\mathcal{G}$-oGG (Functionality 3).

OBSERVE$_{sid}$()
1: if $sid \in \mathfrak{S}$ then  // Restrict caller's session
2:     return $Ob$
```

ants $\mathcal{F}$-wNIZK[$\mathfrak{S}$], Π-G16[$\mathfrak{S}$] that restrict $\mathcal{F}$-wNIZK, Π-G16 to sessions in $\mathfrak{S}$. When queried on other sessions they return $\perp$, see Functionalities 8 and 9.[14]

With these restrictions set up, we show in Fig. 2 how to prove a composition ρ of Π-G16 and Π-EG2 secure, even though Π-G16 requires observability and Π-EG2 cannot tolerate observability. The figure depicts that a real system with both Π-G16[$\mathfrak{S}$] and Π-Enc in the presence of $\mathcal{G}$-oGG[$\varnothing$] ξ-UC-emulates an ideal system with both $\mathcal{F}$-wNIZK[$\mathfrak{S}$] and $\mathcal{F}$-PKE in

[14] This is efficiently implementable. For instance $\mathfrak{S}$ could be the set of strings starting with "G16".

Functionality 8: $\mathcal{F}$-wNIZK$[\mathfrak{S}]$

Parameterized with set $\mathfrak{S}$ of allowed sessions.

```
INIT_sid, PROVE_sid, VERIFY_sid
1: if $sid \notin \mathfrak{S}$ then // Restrict functionality to sessions $sid \in \mathfrak{S}$
2:     return $\perp$
3: continue as in $\mathcal{F}$-wNIZK (Functionality 1).
```

Functionality 9: Π-G16$[\mathfrak{S}]$

Parameterized with set $\mathfrak{S}$ of allowed sessions.

```
INIT_sid, PROVE_sid, VERIFY_sid
1: if $sid \notin \mathfrak{S}$ then // Restrict protocol to sessions $sid \in \mathfrak{S}$
2:     return $\perp$
3: continue as in $\Pi$-G16 (Protocol 1).
```

the presence of $\mathcal{G}$-oGG$[\mathfrak{S}]$ for identity bounds ξ that reject all OBSERVE$_{sid}$ queries for *sid* in $\mathfrak{S}$. In more detail, the following steps are taken in the figure:

– Real to $\mathcal{F}$-PKE Hybrid: We first use the UCGS composition theorem for protocol Π-EG2 emulating $\mathcal{F}$-PKE, which gives us a system with Π-G16$[\mathfrak{S}]$ and $\mathcal{F}$-PKE in the presence of $\mathcal{G}$-oGG$[\varnothing]$. This is possible because $\mathcal{G}$-oGG$[\varnothing]$ behaves like $\mathcal{G}$-GG, without observations, which makes Π-EG2 secure in this setting.

– $\mathcal{F}$-PKE Hybrid to $\mathcal{F}$-PKE Hybrid with observations: We switch on observations by replacing $\mathcal{G}$-oGG$[\varnothing]$ with $\mathcal{G}$-oGG$[\mathfrak{S}]$, which is made possible by Lemma 6 (intuitively, this switch cannot be detected because the environment is ξ-restricted to not test for OBSERVE availability, as are the protocols. The simulator $\mathcal{S}_1$ can be assumed without loss of generality never to call OBSERVE).

– $\mathcal{F}$-PKE Hybrid with observations to $\mathcal{F}$-PKE, $\mathcal{F}$-wNIZK Hybrid with observations: We apply the UCGS composition theorem for protocol Π-G16$[\mathfrak{S}]$ UC-emulating $\mathcal{F}$-wNIZK$[\mathfrak{S}]$ (in the presence of $\mathcal{G}$-oGG$[\mathfrak{S}]$), which is possible because the simulator $\mathcal{S}_2$ is able to ask $\mathcal{G}$-oGG for observations.

7 Conclusion and future work

In this paper, we have established the restricted observable global generic group functionality $\mathcal{G}$-oGG and, as an important application to a widespread SNARK, we have proven Groth16 UC-secure in the $\mathcal{F}$-CRS hybrid model in the presence of $\mathcal{G}$-oGG. We expect the functionality $\mathcal{G}$-oGG to find additional applications, in particular for proving other SNARKs UC-secure, especially ones based on polynomial interactive oracle proofs (PIOPs) [CHM+20, BFS20, CFF+21], such as PLONK [GWC19]. In fact, recent works show that SNARKs obtained from PIOP and the KZG polynomial commitment [KZG10] are already simulation-extractable without modification in the AGM and (programmable) ROM [FFK+23, KPT23, FFR24]. Thus, a natural follow-up question is whether these SNARKs are UC-secure in the presence of $\mathcal{G}$-oGG and (restricted programmable) global random oracle functionalities.

Another exciting research opportunity is to establish a "UC lifting theorem" that allows practitioners to analyze the security of their constructions in the (simpler) game-based generic-group model, and then automatically obtain UC security via lifting. Section 4 already establishes that in spirit, standard GGM proof techniques carry over to the UC setting. Our proof of Groth16 security is a good indicator that the protocol-specific part of the proof mostly boils down to symbolic analysis of polynomials, which is already available from the original paper, or from proofs in the AGM. Establishing formal requirements for a game-based proof to carry over to UC, would be a powerful bridge between game-based "standalone" proofs and UC proofs.

While our paper addresses reuse of the group (multiple protocols using the same group), we leave open the question of a reusable CRS for Groth16, or more generally, the question of reusing (parts of the) CRS across multiple sessions for NIZK in UC. Our Groth16 works in the $\mathcal{F}$-CRS-hybrid model, which means that every session of Groth16 needs its own CRS (which can be "reused" only insofar that parties in the same session can compute multiple proofs from it). The same limitation applies to essentially all existing results on CRS-based NIZK in UC [Gro06, CL06, KZM+15, CsW19, ARS20, BS21, CSW22, LR22b, GKO+23, AGRS23], which also rely on non-reusable, local CRS functionalities. There are multiple ways one can imagine improving upon this situation. First, one could make the same instance of Groth16 available to multiple caller sessions. This means that in a composition, one can use the same instance of Groth16 as a subroutine for multiple protocols. This would also mean that all those subroutines get to share in the same CRS. This is a simple solution, already supported (in spirit) by our security proof of Groth16, but there is a lack of support for this in the UC framework (using the same Π-G16 session in multiple places is not *subroutine respecting*). Second, one could attempt to exchange the local $\mathcal{F}$-CRS for a global CRS functionality $\mathcal{G}$-CRS. However, as is well-known in the literature (e.g., [CDPW07, Section 3]), global CRSs cannot be implemented naively. Third, one may want to share *part* of the CRS (e.g., the part which does not depend on the specific circuit, like the "powers of τ"). There is some work [KMSV21] on this for Groth16. However, it is unclear whether this enables composable analysis. Further research is needed.

We have focused on the strict and observable versions of the global generic group functionalities. Similarly to random oracles [CDG+18], one could envision various levels of programmability for generic groups. While programmability of generic groups is seldomly exploited in game-based proofs (and, to our knowledge, has not been used for NIZK constructions), it is a possibility (e.g., [CDG+22]) and deserves formal UC treatment.

While the generic group model seems to have inherent advantages when it comes to compositional proofs, as discussed in the introduction, the algebraic group model (with oblivious sampling [LPS23]) is the more conservative model (in the sense of restricting the adversary and protocols) in general. An interesting question is whether there is a composable model in the spirit of the AGM that does not restrict the environment from using group elements across sessions.

Finally, we have provided a concrete security analysis of Groth16, giving concrete bounds in Theorem 1. It can be interesting to revisit the tightness of this analysis, especially compared to the game-based setting. However, we are not aware of any GGM-based concrete parameter treatment of Groth16 in the literature, even in the game-based setting. Another interesting direction is to explore what this concrete guarantee means for compositions using Groth16 since concrete security of simulation-based security and of the UC theorem is not well-studied in the literature.

Acknowledgments

We thank Sabine Oechsner for insightful discussions on the connection between this work and State Separating Proofs.

Pooya Farshim was supported in part by EPSRC grant EP/V034065/1. This work was supported by Input Output (iohk.io) through their funding of the Edinburgh ZK-Lab.

for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

ABK$^+$21. M. Abdalla, M. Barbosa, J. Katz, J. Loss, and J. Xu. Algebraic adversaries in the universal composability framework. In *ASIACRYPT 2021, Part III*, vol. 13092 of *LNCS*, pp. 311–341. Springer, Heidelberg, 2021.

AGRS23. B. Abdolmaleki, N. Glaeser, S. Ramacher, and D. Slamanig. Universally composable NIZKs: Circuit-succinct, non-malleable and CRS-updatable. Cryptology ePrint Archive, Report 2023/097, 2023. https://eprint.iacr.org/2023/097.

ARS20. B. Abdolmaleki, S. Ramacher, and D. Slamanig. Lift-and-shift: Obtaining simulation extractable subversion and updatable SNARKs generically. In *ACM CCS 2020*, pp. 1987–2005. ACM Press, 2020.

BBHR19. E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO 2019, Part III*, vol. 11694 of *LNCS*, pp. 701–732. Springer, Heidelberg, 2019.

BCH$^+$20. C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In *TCC 2020, Part III*, vol. 12552 of *LNCS*, pp. 1–30. Springer, Heidelberg, 2020.

BCR$^+$19. E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT 2019, Part I*, vol. 11476 of *LNCS*, pp. 103–128. Springer, Heidelberg, 2019.

BCS16. E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In *TCC 2016-B, Part II*, vol. 9986 of *LNCS*, pp. 31–60. Springer, Heidelberg, 2016.

BDF$^+$18. C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. In *ASIACRYPT 2018, Part III*, vol. 11274 of *LNCS*, pp. 222–249. Springer, Heidelberg, 2018.

BFHK23. B. Bauer, P. Farshim, P. Harasser, and M. Kohlweiss. The uber-knowledge assumption: A bridge to the agm. Cryptology ePrint Archive, Paper 2023/1601, 2023. https://eprint.iacr.org/2023/1601.

BFS20. B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. In *EUROCRYPT 2020, Part I*, vol. 12105 of *LNCS*, pp. 677–706. Springer, Heidelberg, 2020.

BG18. S. Bowe and A. Gabizon. Making groth's zk-SNARK simulation extractable in the random oracle model. Cryptology ePrint Archive, Report 2018/187, 2018. https://eprint.iacr.org/2018/187.

BKSV21. K. Baghery, M. Kohlweiss, J. Siim, and M. Volkhov. Another look at extraction and randomization of groth's zk-SNARK. In *FC 2021, Part I*, vol. 12674 of *LNCS*, pp. 457–475. Springer, Heidelberg, 2021.

BLS01. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *ASIACRYPT 2001*, vol. 2248 of *LNCS*, pp. 514–532. Springer, Heidelberg, 2001.

BMZ19. J. Bartusek, F. Ma, and M. Zhandry. The distinction between fixed and random generators in group-based assumptions. In *CRYPTO 2019, Part II*, vol. 11693 of *LNCS*, pp. 801–830. Springer, Heidelberg, 2019.

BS21. K. Baghery and M. Sedaghat. Tiramisu: Black-box simulation extractable NIZKs in the updatable CRS model. In *CANS 21*, vol. 13099 of *LNCS*, pp. 531–551. Springer, Heidelberg, 2021.

Can01. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pp. 136–145. IEEE Computer Society Press, 2001.

Can20a. R. Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.

Can20b. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Paper 2000/067, 2020. https://eprint.iacr.org/2000/067.

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

CDG$^+$22. B. Chen, Y. Dodis, E. Ghosh, E. Goldin, B. Kesavan, A. Marcedone, and M. E. Mou. Rotatable zero knowledge sets - post compromise secure auditable dictionaries with application to key transparency. In *ASIACRYPT 2022, Part III*, vol. 13793 of *LNCS*, pp. 547–580. Springer, Heidelberg, 2022.

CDPW07. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC 2007*, vol. 4392 of *LNCS*, pp. 61–85. Springer, Heidelberg, 2007.

CF24. A. Chiesa and G. Fenzi. zksnarks in the rom with unconditional uc-security. Cryptology ePrint Archive, Paper 2024/724, 2024. https://eprint.iacr.org/2024/724.

CFF$^+$21. M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodríguez. Lunar: A toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. In *ASIACRYPT 2021, Part III*, vol. 13092 of *LNCS*, pp. 3–33. Springer, Heidelberg, 2021.

CHM$^+$20. A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT 2020, Part I*, vol. 12105 of *LNCS*, pp. 738–768. Springer, Heidelberg, 2020.

CJS14. R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In *ACM CCS 2014*, pp. 597–608. ACM Press, 2014.

CL06. M. Chase and A. Lysyanskaya. On signatures of knowledge. In *CRYPTO 2006*, vol. 4117 of *LNCS*, pp. 78–96. Springer, Heidelberg, 2006.

CNPR22. C. Cremers, M. Naor, S. Paz, and E. Ronen. CHIP and CRISP: Protecting all parties against compromise through identity-binding PAKEs. In *CRYPTO 2022, Part II*, vol. 13508 of *LNCS*, pp. 668–698. Springer, Heidelberg, 2022.

CR03. R. Canetti and T. Rabin. Universal composition with joint state. In *CRYPTO 2003*, vol. 2729 of *LNCS*, pp. 265–281. Springer, Heidelberg, 2003.

CS98. R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO'98*, vol. 1462 of *LNCS*, pp. 13–25. Springer, Heidelberg, 1998.

CsW19. R. Cohen, a. shelat, and D. Wichs. Adaptively secure MPC with sublinear communication complexity. In *CRYPTO 2019, Part II*, vol. 11693 of *LNCS*, pp. 30–60. Springer, Heidelberg, 2019.

CSW22. R. Canetti, P. Sarkar, and X. Wang. Triply adaptive UC NIZK. In *ASIACRYPT 2022, Part II*, vol. 13792 of *LNCS*, pp. 466–495. Springer, Heidelberg, 2022.

DDO$^+$01. A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust non-interactive zero knowledge. In *CRYPTO 2001*, vol. 2139 of *LNCS*, pp. 566–598. Springer, Heidelberg, 2001.

Den02. A. W. Dent. Adapting the weaknesses of the random oracle model to the generic group model. In *ASIACRYPT 2002*, vol. 2501 of *LNCS*, pp. 100–109. Springer, Heidelberg, 2002.

FFK$^+$23. A. Faonio, D. Fiore, M. Kohlweiss, L. Russo, and M. Zajac. From polynomial IOP and commitments to non-malleable zkSNARKs. In *TCC 2023, Part III*, vol. 14371 of *LNCS*, pp. 455–485. Springer, Heidelberg, 2023.

FFR24. A. Faonio, D. Fiore, and L. Russo. Real-world universal zksnarks are non-malleable. Cryptology ePrint Archive, Paper 2024/721, 2024. https://eprint.iacr.org/2024/721.

Fis06. M. Fischlin. Round-optimal composable blind signatures in the common reference string model. In *CRYPTO 2006*, vol. 4117 of *LNCS*, pp. 60–77. Springer, Heidelberg, 2006.

FKL18. G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *CRYPTO 2018, Part II*, vol. 10992 of *LNCS*, pp. 33–62. Springer, Heidelberg, 2018.

FKMV12. S. Faust, M. Kohlweiss, G. A. Marson, and D. Venturi. On the non-malleability of the Fiat-Shamir transform. In *INDOCRYPT 2012*, vol. 7668 of *LNCS*, pp. 60–79. Springer, Heidelberg, 2012.

GKO$^+$23. C. Ganesh, Y. Kondi, C. Orlandi, M. Pancholi, A. Takahashi, and D. Tschudi. Witness-succinct universally-composable SNARKs. In *EUROCRYPT 2023, Part II*, vol. 14005 of *LNCS*, pp. 315–346. Springer, Heidelberg, 2023.

GM17. J. Groth and M. Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In *CRYPTO 2017, Part II*, vol. 10402 of *LNCS*, pp. 581–612. Springer, Heidelberg, 2017.

Gro06. J. Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *ASIACRYPT 2006*, vol. 4284 of *LNCS*, pp. 444–459. Springer, Heidelberg, 2006.

Gro16. J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016, Part II*, vol. 9666 of *LNCS*, pp. 305–326. Springer, Heidelberg, 2016.

GWC19. A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

HS15. D. Hofheinz and V. Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, 2015.

JKK14. S. Jarecki, A. Kiayias, and H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT 2014, Part II*, vol. 8874 of *LNCS*, pp. 233–253. Springer, Heidelberg, 2014.

JR10. T. Jager and A. Rupp. The semi-generic group model and applications to pairing-based cryptography. In *ASIACRYPT 2010*, vol. 6477 of *LNCS*, pp. 539–556. Springer, Heidelberg, 2010.

Kil92. J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, pp. 723–732. ACM Press, 1992.

KKK21. T. Kerber, A. Kiayias, and M. Kohlweiss. Composition with knowledge assumptions. In *CRYPTO 2021, Part IV*, vol. 12828 of *LNCS*, pp. 364–393, Virtual Event, 2021. Springer, Heidelberg.

KMS$^+$16. A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pp. 839–858. IEEE Computer Society Press, 2016.

KMSV21. M. Kohlweiss, M. Maller, J. Siim, and M. Volkhov. Snarky ceremonies. In *ASIACRYPT 2021, Part III*, vol. 13092 of *LNCS*, pp. 98–127. Springer, Heidelberg, 2021.

KPT23. M. Kohlweiss, M. Pancholi, and A. Takahashi. How to compile polynomial IOP into simulation-extractable SNARKs: A modular approach. In *TCC 2023, Part III*, vol. 14371 of *LNCS*, pp. 486–512. Springer, Heidelberg, 2023.

Küs06. R. Küsters. Simulation-based security with inexhaustible interactive Turing machines. Cryptology ePrint Archive, Report 2006/151, 2006. https://eprint.iacr.org/2006/151.

KZG10. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, vol. 6477 of *LNCS*, pp. 177–194. Springer, Heidelberg, 2010.

KZM$^+$15. A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, a. shelat, and E. Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. https://eprint.iacr.org/2015/1093.

LPS23. H. Lipmaa, R. Parisella, and J. Siim. Algebraic group model with oblivious sampling. In *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part IV*, vol. 14372 of *Lecture Notes in Computer Science*, pp. 363–392. Springer, 2023.

LR22a. A. Lysyanskaya and L. N. Rosenbloom. Efficient and universally composable non-interactive zero-knowledge proofs of knowledge with security against adaptive corruptions. Cryptology ePrint Archive, Report 2022/1484, 2022. https://eprint.iacr.org/2022/1484.

LR22b. A. Lysyanskaya and L. N. Rosenbloom. Universally composable Σ-protocols in the global random-oracle model. In *TCC 2022, Part I*, vol. 13747 of *LNCS*, pp. 203–233. Springer, Heidelberg, 2022.

Mau05. U. M. Maurer. Abstract models of computation in cryptography (invited paper). In *10th IMA International Conference on Cryptography and Coding*, vol. 3796 of *LNCS*, pp. 1–12. Springer, Heidelberg, 2005.

Mau10. U. Maurer. Constructive cryptography - a primer (invited paper). In *FC 2010*, vol. 6052 of *LNCS*, p. 1. Springer, Heidelberg, 2010.

Mic00. S. Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

Nec94. V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

PS16. D. Pointcheval and O. Sanders. Short randomizable signatures. In *CT-RSA 2016*, vol. 9610 of *LNCS*, pp. 111–126. Springer, Heidelberg, 2016.

Sah99. A. Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th FOCS*, pp. 543–553. IEEE Computer Society Press, 1999.

Sho97. V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97*, vol. 1233 of *LNCS*, pp. 256–266. Springer, Heidelberg, 1997.

Sma01. N. P. Smart. The exact security of ECIES in the generic group model. In *8th IMA International Conference on Cryptography and Coding*, vol. 2260 of *LNCS*, pp. 73–84. Springer, Heidelberg, 2001.

SPMS02. J. Stern, D. Pointcheval, J. Malone-Lee, and N. P. Smart. Flaws in applying proof methodologies to signature schemes. In *CRYPTO 2002*, vol. 2442 of *LNCS*, pp. 93–110. Springer, Heidelberg, 2002.

Zha22. M. Zhandry. To label, or not to label (in generic groups). In *CRYPTO 2022, Part III*, vol. 13509 of *LNCS*, pp. 66–96. Springer, Heidelberg, 2022.

ZZK22. C. Zhang, H.-S. Zhou, and J. Katz. An analysis of the algebraic group model. In *ASIACRYPT 2022, Part IV*, vol. 13794 of *LNCS*, pp. 310–322. Springer, Heidelberg, 2022.

A Related work on UC security and SNARKs

Property-based definitions and UC-NIZK A number of works study the relation between property-based and UC definitions for NIZKs. The first key property is *straightline* simulation and extraction: since an UC experiment by definition does not allow the simulator to rewind the environment, one must construct a simulator that performs both ZK simulation and witness extraction straightline. Another critical property is *non-malleability*, which is often referred to as *simulation-extractability (SE)* in the context of NIZK proof systems [Sah99, DDO+01, Gro06, FKMV12]. The SE property can be dissected into two flavors: 1. "weak" SE in the sense that an adversary cannot forge a proof without knowing witness for at least those statements that have never been queried to the simulation oracle, and 2. "strong" SE that prevents an adversary from mauling proof for a statement that has already been queried to the simulation oracle. Groth [Gro06, Theorem 20] shows that CRS-based (straightline) strong SE NIZK UC-realizes the NIZK functionality in the CRS-hybrid model. Chase and Lysyanskaya [CL06, Theorem 2.2] show the equivalence between strong SE and UC security for signature of knowledge. The recent work of Chiesa and Fenzi [CF24] presents "UC-friendly" property-based security definitions for RO-based NIZK and proves that any NIZK satisfying these properties in the ROM is UC-secure in the presence of restricted observable and programmable global random oracle. Their result implies that purely RO-based SNARKs such as those of Kilian–Micali [Kil92, Mic00] and Interactive Oracle Proof compiled with the Merkle-tree e.g. [BCS16, BCR+19, BBHR19] are already UC secure. As observed by Kosba et al. [KZM+15, KMS+16] weak SE suffices for a typical UC application. It is well known that Groth16 in its original form is not strong SE because its proof is re-randomizable. Baghery et al. [BKSV21] show Groth16 still satisfies weak SE in the AGM. Groth and Maller [GM17] presented a modified version of Groth16 which satisfies strong SE under knowledge assumptions. Bowe and Gabizon [BG18] also showed a way to make Groth16 strong SE in the ROM and GGM.

UC-lifting Compiler There exist several generic compilers in the literature that lift NIZK into a UC-secure one under various assumptions. However, all these approaches incur overhead and hence are not ideal from a practical perspective. Kosba et al. [KZM+15, Theorem 2] give a generic compiler (the CØCØ transformation) that turns any NIZK into a scheme that UC-realizes $\mathcal{F}$-$\texttt{NIZK}$ in the ($\mathcal{F}$-$\texttt{CRS}$)-hybrid model using a CPA-secure PKE and a PRF. Although several follow-up works appeared to reduce the overhead incurred by the CØCØ-style transformation [ARS20, BS21, AGRS23], the transformation requires the prover to encrypt the witness, and thus the construction loses witness succinctness. Lysyanskaya et al. [LR22b, Theorem 3] show a generic compiler that converts any Σ-protocols into a NIZK that GUC-realizes $\mathcal{F}$-$\texttt{NIZK}$ in the ($\mathcal{G}$-$\texttt{RO}$, $\mathcal{F}$-$\texttt{CRS}$)-hybrid model. Ganesh et al. [GKO+23, Theorem 1] present a compiler that turns CRS-based simulation-extractable NIZK (where extraction is non-black-box) into a scheme that UC-realizes $\mathcal{F}$-$\texttt{NIZK}$ the ($\mathcal{G}$-$\texttt{RO}$, $\mathcal{F}$-$\texttt{CRS}$)-hybrid model. Their method preserves succinctness, but still involves "compilation" and a straightline extraction enabled by additional proof-of-work reminiscent of Fischlin's transformation [Fis06], which seems too high for practitioners.

B FAQ

This section collects questions and answers from reviews and internal discussions.

Is there a concrete example of an attack that the UC-AGM excludes? Suppose we have a protocol ρ with access to two communication channel functionalities $\mathcal{F}$-$\texttt{Ch1}$, $\mathcal{F}$-$\texttt{Ch2}$. Now consider the following behavior of a party of ρ:

– ρ generates a random secret $sk \xleftarrow{\$} \mathbb{Z}_p$ and corresponding public key $pk = g^{sk}$.

– ρ sends pk over $\mathcal{F}$-Ch1 to all other parties.

– If ρ receives pk over $\mathcal{F}$-Ch2, then it reveals its secret key sk.

Clearly, ρ is insecure (assuming knowledge of sk breaks ρ) and the attack is to just send pk via $\mathcal{F}$-Ch2 to some honest party to learn their secret key.

However, in the UC-AGM, this attack is not allowed and the protocol above would be considered secure. This is because the adversary cannot send pk over $\mathcal{F}$-Ch2, since it does not know the discrete logarithm of pk w.r.t. the AGM-basis of $\mathcal{F}$-Ch2. When ρ sends pk over $\mathcal{F}$-Ch1 to the adversary, the element pk is added to the AGM-basis of $\mathcal{F}$-Ch1 (which enables the adversary to send pk via $\mathcal{F}$-Ch1 with the trivial representation), but *not* to the AGM-basis of $\mathcal{F}$-Ch2. This is a feature because otherwise, sibling functionalities could add group elements to each other's AGM-bases, which makes certain security proofs impossible. However, it demonstrates how certain scenarios cannot be meaningfully modeled in the UC-AGM.

We note that [ABK$^+$21] already discusses inherent shortcomings of the AGM when it comes to composition, noting that cross-session attacks are problematic and that their paper wants to explore those limits. Our example is not supposed to falsify [ABK$^+$21] (indeed, we have no reason to believe their theorems are wrong), but rather to further illustrate those limitations.

How do you model global functionalities in UC? Is it related to GUC/EUC? In contrast to GUC/EUC [CDPW07], we work in the plain UC model, which *does* allow modeling global functionalities (but does not, by itself, provide a composition theorem that works in the presence of global functionalities). We follow the UCGS [BCH$^+$20] formalism, which uses plain UC but adds a composition theorem ([BCH$^+$20, Theorem 3.5]) that works in the presence of global subroutines such as $\mathcal{G}$-oGG. In that setting, the environment gets direct access to the global subroutine just as described in our paper (e.g., Fig. 1). The similarity of UCGS to EUC [CDPW07] is not accidental: UCGS solves the same issue as EUC. On a high level, the two models are essentially the same, but EUC is based on an old version of UC, while UCGS makes black-box use of the most recent version of UC.

In this result, can the Groth16 CRS be reused? In our current formulation, the CRS ($\mathcal{F}$-CRS) is local to each Groth16 (Π-G16) instance and cannot be accessed by other instances or protocols (though it should be stressed that it *can* be accessed by the adversary). As a consequence, the CRS cannot be reused for any purposes by other Groth16 instances or protocols. This is standard modeling for UC-NIZK protocols [Gro06, CL06, KZM$^+$15, CsW19, ARS20, BS21, CSW22, LR22b, GKO$^+$23, AGRS23].

Can I use Groth16 in UC to prove something about a commitment using the same group? No. The issue is that if we model the group that Groth16 and the commitment uses as a generic group, then there is no way to design an efficient circuit (or QAP) to verify the commitment. This is because for generic groups, there is no circuit that can evaluate/check group operations. The situation is similar to proving statements involving a hash function modeled as the random oracle. Even in a game-based setting, the security guarantee is unclear if the soundness of a proof system relies on GGM while the statement contains concrete group descriptions. We believe this is more of a question about the GGM in general, but not an issue caused by "composition" as in the UC terminology. In this regard, the AGM is the more useful model, given that in the AGM, group operations *can* be done in a ZK circuit.

In $\mathcal{G}$-oGG, what session does $\tau_i(0)$ belong to? What about $\tau_i(1)$? The group's neutral element $\tau_i(0)$ is shared among all sessions in the sense that operations with it do not trigger observability. This contributes to unobservability being "closed" in $\mathcal{G}$-oSG, meaning that computing $g_{sid} - g_{sid}$ does not become observable.

Functionality 10: $\mathcal{F}$-`Setup` / $\mathcal{G}$-`Setup`

```
EVAL_sid(x)
1: $y \leftarrow [\ldots]$ // e.g., $y = \mathcal{H}(x)$ (ROM) or $y = \tau(\tau^{-1}(x_1) + \tau^{-1}(x_2))$ (GGM)
2: return $y$
```

In contrast to that, $\tau_i(1)$ is the canonical generator of the group. Whenever any protocol uses $\tau_i(1)$, those operations are observable. Using $\tau_i(1)$ corresponds to having a constant term in an element's polynomial representation (e.g., computing "$\tau_i(h) = 5 \cdot \tau_i(1) + \tau_i(g_{sid})$" results in $\mathsf{R}_i[h] = \{5 + \mathsf{X}_{g_{sid}}\} \not\subseteq \mathsf{Legal}_{sid}$).

C An overview of observability in global functionalities

In this section, we revisit the ideas behind observable global functionalities, motivating their general design. As an example, we consider a NIZKPoK scheme π, where the strategy for extraction is to observe queries that a malicious prover makes. This discussion applies to both the case where the observations are (i) random oracle queries, or (ii) generic group operation queries. To unify both, we will generically talk about a "setup" functionality $\mathcal{G}$-`Setup` with some interface EVAL, where EVAL may either return random oracle images or the result of generic group operations.

We start by looking at the situation for a local functionality $\mathcal{F}$-`Setup`, compare it to its global equivalent $\mathcal{G}$-`Setup`, then discuss straw man approaches for enabling observations, and finally look at the proper observable functionality $\mathcal{G}$-`oSetup`.

Observation for local functionalities. When we say "local functionality", we mean that an instance of this functionality exists independently for every instance of a protocol using it. This models that every protocol (session) gets its own independent random oracle or its own independent generic group. In UC, this is formalized by proving π secure in the $\mathcal{F}$-`Setup` hybrid setting, as in Fig. 3. This is arguably not a faithful modeling of the world we live in, where many protocols share the same instance of, say, the hash function SHA-3, or the bilinear group BLS12-381.

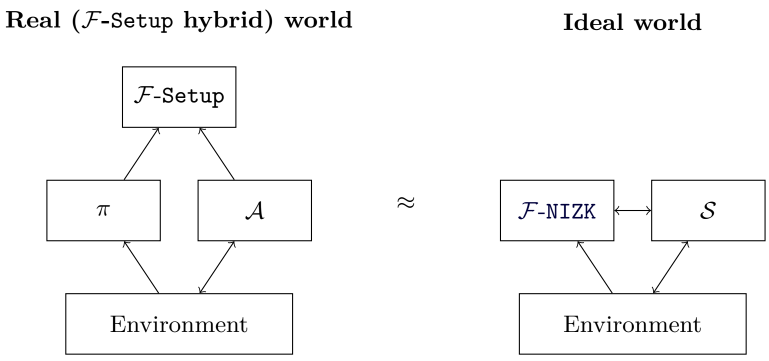

Fig. 3: An illustration of the $\mathcal{F}$-`Setup` hybrid setting and the ideal world for proving that π UC-realizes $\mathcal{F}$-`NIZK`. We omit the dummy parties for $\mathcal{F}$-`NIZK`.

For local functionalities, the situation for observability is very simple: In the real world, $\mathcal{F}$-`Setup` can only be accessed by the protocol π and the (local) adversary $\mathcal{A}$. In the ideal world, we can think of $\mathcal{F}$-`Setup` as being simulated by $\mathcal{S}$: if $\mathcal{A}$ in the real world wants to access $\mathcal{F}$-`Setup`, then $\mathcal{S}$ can simply execute this access in its head. $\mathcal{S}$ may even deviate from the behavior of $\mathcal{F}$-`Setup` (e.g., program the random oracle), as long as this is undetectable to the real-world adversary $\mathcal{A}$ and the environment. Regarding observability, this situation is quite comfortable for $\mathcal{S}$. As all access to (the simulated version of) $\mathcal{F}$-`Setup` goes through $\mathcal{S}$, it gets to observe all queries.

The (observation) issues with the global functionality. If we switch our view from the local $\mathcal{F}$-`Setup` to the global $\mathcal{G}$-`Setup`, the situation changes. $\mathcal{G}$-`Setup` being a global functionality means that it is (potentially) *shared* among multiple instances of multiple protocols. This is modeled by allowing the environment (which represents other instances of other protocols running concurrently with π) direct access to $\mathcal{G}$-`Setup` (cf. Fig. 4). More specifically, the protocol π and the adversary $\mathcal{A}$ get to make $\mathcal{G}$-`Setup` queries for their session *sid* (i.e. calls to EVAL_{sid}), while the environment gets to make queries for all other sessions $sid' \neq sid$. Note that the environment can *indirectly* query EVAL_{sid} for the protocol's session *sid* simply by asking the adversary $\mathcal{A}$ to make the query.

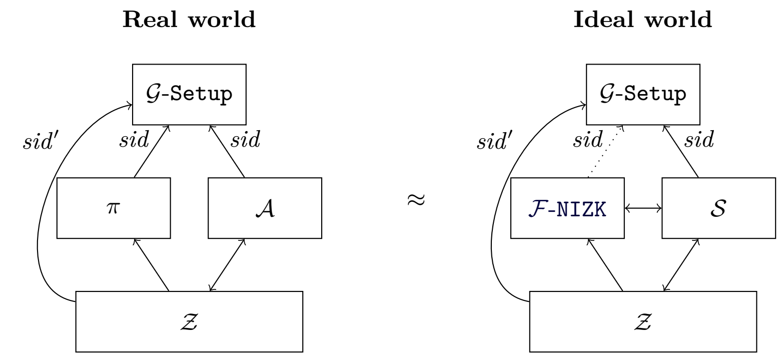

Fig. 4: An illustration of the real and ideal world setting for proving that π UC-realizes $\mathcal{F}$-`NIZK` in the presence of the global $\mathcal{G}$-`Setup`. We omit the dummy parties for $\mathcal{F}$-`NIZK`.

We find a similar situation in the ideal world (cf. Fig. 4). The environment still gets direct access to $\mathcal{G}$-`Setup` for $sid' \neq sid$ (as clearly, other protocols should not lose access to $\mathcal{G}$-`Setup` just because we replace our protocol by its ideal counterpart). The ideal functionality $\mathcal{F}$-`NIZK` and the simulator $\mathcal{S}$ can access $\mathcal{G}$-`Setup` for their session *sid*. Note that $\mathcal{F}$-`NIZK`, while it technically could access $\mathcal{G}$-`Setup`, does not actually do so.

In contrast to the local $\mathcal{F}$-`Setup`, the simulator $\mathcal{S}$ does not fully control $\mathcal{G}$-`Setup`. This makes sense from a composability standpoint: say $\mathcal{G}$-`Setup` is a shared random oracle between two protocols, each of which is secure only if their respective simulator programs the random oracle hash of 0—clearly, we would run into conflicts when proving security of the composed protocol.

Furthermore, $\mathcal{S}$ does not get to see all accesses to $\mathcal{G}$-`Setup`. While accesses made by the adversary $\mathcal{A}$ for session *sid* in the real world are still conceptually visible to $\mathcal{S}$ (given that $\mathcal{S}$ can internally simulate $\mathcal{A}$), accesses made by the environment for sessions sid' happen without involvement of $\mathcal{S}$. This means that the environment can easily circumvent its queries from being observed by $\mathcal{S}$. As a consequence, for example, the environment can honestly compute a NIZK proof p by querying the random oracle in session sid' and then submit it for verification. Verification will succeed in the real world, but because the simulator was not able to observe the environment's queries, it cannot extract a witness from proof p, making verification fail.

To fix this issue, we need to give $\mathcal{S}$ the ability to observe the queries that the environment makes to $\mathcal{G}$-`Setup` (at least the ones that are relevant to proofs in session *sid*).

Straw man 1: just make all queries observable to everyone. To make the environment's queries observable, we could just augment $\mathcal{G}$-`Setup` with an interface Observe that simply outputs the list of all x ever queried to Eval. This would enable our NIZK simulator to extract from proofs generated by the environment in other sessions. However, while full observability is a legitimate setting to consider, it means that *everyone* is able to observe *all* queries. In particular, the environment can extract honest parties' proofs in the real world (by observing their $\mathcal{G}$-`Setup` queries and then using the same extraction strategy the simulator would). Proofs created by the simulator in the ideal world can generally not be extracted from, which allows the environment to distinguish the two worlds.

This highlights that observability must not be absolute: for some properties, like zero-knowledge, we want to hide some $\mathcal{G}$-`Setup` queries, namely those made by the honest parties in the real world and by the simulator in the ideal world.

Straw man 2: only record $\mathcal{G}$-`Setup` queries made by the environment. As alluded to above, for the NIZKPoK application, the dream scenario would be that

- Our simulator $\mathcal{S}$ sees all $\mathcal{G}$-`Setup` queries made by the environment (this enables proof of knowledge extraction).
- The environment does not see the queries of honest parties or the simulator (this enables zero-knowledge simulation).

So it might be tempting to just define $\mathcal{G}$-`Setup` exactly so that these two conditions hold. So it would be nice if we could make Observe just output the list of all queries made by the environment. However, in UC, there is no reasonable way for $\mathcal{G}$-`Setup` to express an "if the caller is the environment" check. This is for good reason: the environment is supposed to be replaceable with arbitrary protocols, so $\mathcal{G}$-`Setup` cannot treat the environment differently from the honest/corrupted protocol parties that it represents.

We could look at Fig. 4 and notice that queries made by the environment are w.r.t. sessions $sid' \neq sid$. So we could potentially define $\mathcal{G}$-`Setup` to make queries in sessions sid' observable, but queries in session *sid* unobservable. This would result in, effectively, a parameterized functionality $\mathcal{G}$-`Setup`$[sid]$ that gives the specific session *sid* preferential treatment. This is not desirable (e.g., we would not be able to compose two protocols sharing the same $\mathcal{G}$-`Setup`, if they are proven secure only w.r.t. $\mathcal{G}$-`Setup`$[sid_1]$ and $\mathcal{G}$-`Setup`$[sid_2]$, respectively).

And while the solution will indeed revolve around session IDs, its treatment of sessions will be "symmetric", in a manner of speaking, and not single out specific sessions. This means that we want to set up a universal rule that applies to all session IDs equally, so that we do not run into issues where every protocol session *sid* needs its own version of the global $\mathcal{G}$-`Setup`$[sid]$.

Observable global functionalities with domain separation. What turned out to be the best way to model observability is through domain separation. This is intuitively reasonable, given that we are ultimately trying to share a resource $\mathcal{G}$-`Setup` between multiple protocols. Domain separation gives a way of "dividing up" the resource.

The general idea here is that honest parties and simulators are expected to *respect domain separation*, i.e. they only query $\text{Eval}_{sid}(x)$ for inputs x that *belong to* their session *sid*.

The exact notion of an input x *belonging to* a session *sid* depends on the functionality. For random oracles, the standard notion is that we expect the hash preimage x to carry the session ID *sid* as a prefix, i.e. $x = (sid, x')$ for some $x' \in \{0,1\}^*$. For generic groups, roughly speaking, every session is associated with a set of generators, and we expect the input group elements $(g_1, g_2) = x$ to be derived (only) from the generators of session *sid*.

Functionality 11: $\mathcal{G}$-oSetup

State: Ob, an initially empty list of observations.

```
EVAL_sid(x)   // caller's session: sid
1: if $x$ does not belong to $sid$ then
2:     $Ob \leftarrow Ob : [x]$
3: $y \leftarrow [\dots]$
4: return $y$

OBSERVE_sid()
5: return $Ob$
```

Functionality 12: $\mathcal{G}$-oRO

$\mathcal{G}$-oRO is parameterized by finite set $S \in \{0,1\}^*$. State: Ob, an initially empty list of observations, an initially empty table T

```
EVAL_sid(x)   // caller's session: sid
1: if $x$ is not of the form $(sid, x')$ then
2:     $Ob \leftarrow Ob : [x]$
3: if $T[x] = \perp$ then $T[x] \xleftarrow{\$} S$
4: return $T[x]$

OBSERVE_sid()
5: return $Ob$
```

As shown in Functionality 11, queries to $\mathcal{G}$-oSetup that *respect* domain separation are unobservable. This gives protocol parties and the simulator an easy way to avoid being observed. Queries that *violate* domain separation are observable. This gives the simulator a way to observe (some) queries made by the environment.

As a concrete example for $\mathcal{G}$-oSetup, see Functionality 12 for the global observable random oracle functionality $\mathcal{G}$-oRO as found in the literature [CJS14, CDG+18].

In Table 1, we give an overview of different kinds of queries to $\mathcal{G}$-oSetup and their observability in the NIZK use case.

Query	**Observability**
Env queries $\text{EVAL}_{sid'}(x)$	Query **observable** (x does not belong to sid'). Helps $\mathcal{S}$ extract.
Env queries $\text{EVAL}_{sid'}(x')$	Query **unobservable** (x' belongs to sid'), but x' should be irrelevant for (extraction) task of $\mathcal{S}$.
$\mathcal{A}$ queries $\text{EVAL}_{sid}(x)$	Query **unobservable** in real world, but $\mathcal{A}$ is taken over/simulated by $\mathcal{S}$ in the ideal world, so **$\mathcal{S}$ sees all queries that $\mathcal{A}$ would make** anyway. Helps $\mathcal{S}$ extract.
$\mathcal{S}$ queries $\text{EVAL}_{sid}(x)$	Query **unobservable**. Allows $\mathcal{S}$ to produce simulated NIZK proofs.
$\mathcal{S}$ queries $\text{EVAL}_{sid}(x')$	Query **observable**. Should usually be avoided by $\mathcal{S}$.
$\mathcal{A}$ queries $\text{EVAL}_{sid}(x')$	Query **observable**. Note that Env can query for x' without being observed, which can be assumed to be the better distinguishing strategy for Env/$\mathcal{A}$.
Env queries $\text{EVAL}_{sid}(\cdot)$	**Not allowed** by UC model, environment cannot query on behalf of target session *sid*. Query is implicitly rejected/ignored by $\mathcal{G}$-oSetup.
$\mathcal{A}$ queries $\text{EVAL}_{sid'}(\cdot)$	**Not allowed** by UC model, $\mathcal{A}$ has session *sid*. Query is implicitly rejected/ignored by $\mathcal{G}$-oSetup.
$\mathcal{S}$ queries $\text{EVAL}_{sid'}(\cdot)$	**Not allowed** by UC model, $\mathcal{S}$ has session *sid*. Query is implicitly rejected/ignored by $\mathcal{G}$-oSetup.

Table 1: Overview of types of queries for $\mathcal{G}$-oSetup when proving something about NIZKPoK protocol π in session *sid* (as depicted in Fig. 4, but with $\mathcal{G}$-Setup replaced with $\mathcal{G}$-oSetup). Notation: x belongs to *sid* and x' belongs to $sid' \neq sid$.

Note that $\mathcal{S}$ does not get to see *all* queries (which is in contrast to security proofs that assume that the simulator gets full control over $\mathcal{F}$-`Setup`). The simulator $\mathcal{S}$ can observe all queries for x that belong to session *sid* (this can be checked via Table 1). But the environment can make queries for x' that belong to other sessions sid'. For security proofs of protocols using $\mathcal{G}$-`oSetup`, one must argue that those queries are *irrelevant* for $\mathcal{S}$'s task of simulating the protocol in session *sid*. In the random oracle case, this is quite straightforward: for a protocol that is written to work with queries $\mathcal{H}(sid, \cdot)$, queries to $\mathcal{H}(sid', \cdot)$ are completely irrelevant (indeed, this kind of domain-separation by prefixing is a folklore strategy to duplicate a single random oracle into multiple *independent* ones). In the generic group case, things are somewhat more complicated, but the general argument is that the protocol makes checks with respect to some generator g_{sid}, which should likely fail if the thing that is checked contains some independent random generator $g_{sid'}$. For a concrete example, see the proof of Claim 5.

Overall, domain separation as in $\mathcal{G}$-`oSetup` allows us to have our cake and eat it, too, i.e. give our simulator access to relevant observations, while still working over a shared resource.

Another (informal) way of looking at domain separation in the context of composition is as follows. Consider a domain separation respecting protocol Π w.r.t. $\mathcal{G}$-`oSetup`. Domain separation basically means that we can compose Π with other protocols Π' that also respect domain separation, as, intuitively, those protocols do not interfere with our protocol (e.g., their hashes have a different prefix, or their group uses a different generator). We then limit the damage that a domain-separation-*violating* protocol Π' can do to our protocol Π, by making the queries of Π' observable. We further limit the damage that adversaries (or simulators) for Π' can do, by hiding the (domain-separation-respecting) queries that Π makes from them.

D Failed attempts at the $\mathcal{G}$-`oGG` functionality

In this section, we discuss earlier attempts at modeling $\mathcal{G}$-`oGG`, motivating our final polynomial-based observation rule of Functionality 3. To simplify this discussion, we concentrate on the generic group model without efficient pairing. Furthermore, instead of allowing sessions to set up multiple generators via Touch, we will simply assume that each session *sid* has a single random generator h_{sid}. Functionality 13 shows a version of Functionality 3 in this simplified setting. Note that in the absence of a pairing, all the polynomials in $\mathsf{R}[h]$ are simply degree 1 polynomials, and checking observability (Line 10) boils down to checking whether the polynomial f resulting from the operation is of the form $a \cdot \mathsf{X}_{sid}$ for some $a \in \mathbb{Z}_p$.

First attempt: Simple $sid + sid = sid$ infection mechanism. Our first attempt (Functionality 14) at $\mathcal{G}$-`oGG` is quite a bit simpler than the final version (Functionality 13). Instead of keeping track of polynomials for each group element, we have a set $V_{sid} \subseteq S$ for each session, which initially contains the generator h_{sid} of session *sid*. When a group operation is performed between two elements $g_1, g_2 \in V_{sid}$, we add the resulting group element to V_{sid}. Operations that involve group elements not in V_{sid} are logged as observable actions.

At first glance, this seems like a reasonable approach. If honest parties in session *sid* only do operations on elements derived from h_{sid}, those operations are unobservable, intuitively enabling properties like zero-knowledge. All other operations are observable with overwhelming probability (e.g., when a party from another session $sid' \neq sid$ uses some $g_1 \in V_{sid}$), enabling properties like proof of knowledge.

The issue with Functionality 14 is subtle. As it turns out, the fact that V_{sid} contains exactly all (intermediate) computation results made by honest protocol parties in session *sid*

Functionality 13: $\mathcal{G}$-oGG without pairing and TOUCH

$\mathcal{G}$-oGG maintains the following state:

- $\tau : \mathbb{Z}_p \to S$ a random encoding function
- $\mathsf{R}[h]$ for $h \in S$ initially empty sets of polynomials
- Ob initially empty list of observable actions.

```
GETGEN_sid()
1: if $h_{sid} = \perp$ then
2:     $h_{sid} \xleftarrow{\$} S$
3:     Initialize fresh variable $\mathsf{X}_{sid}$
4:     $\mathsf{R}[h_{sid}] \leftarrow \mathsf{R}[h_{sid}] \cup \{\mathsf{X}_{sid}\}$
5: return $h_{sid}$

OBSERVE_sid()
6: return $Ob$

OP_sid($g_1, g_2, a_1, a_2$)
7: assert $(g_1, g_2, a_1, a_2) \in S^2 \times \mathbb{Z}_p^2$
8: $h \leftarrow \tau(a_1\tau^{-1}(g_1) + a_2\tau^{-1}(g_2))$
9: $\mathsf{R}[h] \leftarrow \mathsf{R}[h] \cup (a_1\mathsf{R}[g_1] + a_2\mathsf{R}[g_2])$
10: if $\exists \mathsf{f} \in \mathsf{R}[h] : \mathsf{f} \notin \{a \cdot \mathsf{X}_{sid} \mid a \in \mathbb{Z}_p\}$ then
11:     $Ob \leftarrow Ob : [(\text{OP}, g_1, g_2, a_1, a_2, h)]$
12: return $h$
```

Functionality 14: Failed attempt 1 for $\mathcal{G}$-oGG

$\mathcal{G}$-oGG maintains the following state:

- $\tau : \mathbb{Z}_p \to S$ a random encoding function
- $V_{sid} \subseteq S$ initially empty sets of group elements belonging to session *sid*
- Ob initially empty list of observable actions

```
GETGEN_sid()
1: if $h_{sid} = \perp$ then
2:     $h_{sid} \xleftarrow{\$} S$
3:     $V_{sid} \leftarrow \{h_{sid}\}$
4: return $h_{sid}$

OBSERVE_sid()
5: return $Ob$

OP_sid($g_1, g_2, a_1, a_2$)
6: assert $(g_1, g_2, a_1, a_2) \in S^2 \times \mathbb{Z}_p^2$
7: $h \leftarrow \tau(a_1\tau^{-1}(g_1) + a_2\tau^{-1}(g_2))$
8: if $g_1, g_2 \in V_{sid}$ then
9:     $V_{sid} \leftarrow V_{sid} \cup \{h\}$
10: else
11:     $Ob \leftarrow Ob : [(\text{OP}, g_1, g_2, a_1, a_2, h)]$
12: return $h$
```

is an issue. Note that the environment may learn something about the contents of V_{sid} by essentially checking whether certain group operations are observable. As a result, Functionality 14 reveals too much information about the internal computations of session *sid* parties, interfering with properties such as zero-knowledge.

As an illustration of this issue, consider the Groth16 protocol (Protocol 1). In the real world, when computing the proof element A within $\textsc{Prove}_{sid}(x, w)$, the honest prover computes, among others, the intermediate result $g_{\text{intermediate}} = \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid}$, where $(a_i)_{i=0}^{\ell} = x$ is the public input, $(a_i)_{i=\ell+1}^{m} = w$ is the witness, and $u_i(x)$ are (constant) QAP polynomials. It is to be noted that this intermediate result is deterministically computed by the honest prover exactly like this. As a result of the Functionality 14 rules, $g_{\text{intermediate}}$ is added to V_{sid}. In the ideal world, the simulator *cannot* reproduce the same intermediate result, as it does not know the witness. Indeed, the simulator (Simulator 1) simply computes a random A. As a consequence, if the environment is able to check whether $g_{\text{intermediate}} \in V_{sid}$, it can distinguish the real world (where $g_{\text{intermediate}} \in V_{sid}$) from the ideal world (where $g_{\text{intermediate}} \notin V_{sid}$).

This check is indeed easily implemented: the environment would simply compute the element $g_{\text{intermediate}} = \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid}$ itself via $\mathcal{G}$-oGG in some session $sid' \neq sid$. The environment's operations are observable, but do not change V_{sid} for the target session *sid*. It then queries $\textsc{Verify}_{sid}(x, \pi = (A = h_{sid}, B = h_{sid}, C = g_{\text{intermediate}}))$ on some honest verifier. In the real world, the group operations within $\textsc{Verify}$ will be unobservable, as they only involve elements $A, B, C \in V_{sid}$. In the ideal world, some verification operations (involving C) will be observable, as $C \notin V_{sid}$.[15] As a result, the environment can distinguish the two worlds, just using leakage of $\mathcal{G}$-oGG operations via V_{sid} in the Functionality 14 setting.

Overall, Functionality 14 is intuitively unsatisfying because we do not want $\mathcal{G}$-oGG to leak significant information about internal intermediate computations to other sessions. With our final polynomial-based observation rule (Functionality 13), when the environment tries to mount the attack above, the $\textsc{Verify}$ operations on $C = g_{\text{intermediate}} = \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid}$ will be unobservable in both the real and the ideal world, as the rules of Functionality 13 do not care whether $g_{\text{intermediate}}$ has already been computed by the honest prover or whether the environment (session sid') was the one to compute it for the first time. It just looks at $g_{\text{intermediate}}$ symbolically, using the element's polynomial representation $\sum_{i=0}^{m} a_i u_i(x) \cdot \mathsf{X}_{sid}$, and determines that it belongs to session *sid*, no matter who computed it.

Second attempt: Decoupling V_{sid} maintenance from the caller session. The first attempt suffers from an attack where the environment is able to check whether a certain intermediate result $g_{\text{intermediate}}$ has been computed by the honest prover. One potential way to fix this is by making sure that when the environment computes $g_{\text{intermediate}}$ for its attack, then $g_{\text{intermediate}}$ is also added to V_{sid} (at which point the attack fails because $g_{\text{intermediate}} \in V_{sid}$ in both the real and ideal world). In other words, we decouple the maintenance of V_{sid} from the caller's session, making it so that when the environment calls $\textsc{Op}_{sid'}$ in session sid', the result is still added to V_{sid} when appropriate.

This is formalized in Functionality 15. $\textsc{Op}$ now maintains $V_{sid'}$ for all sid' (including $sid' = sid$), and then uses the V_{sid} corresponding to the caller's session to decide whether the operation is observable. This approach fulfills the general requirements (honest parties' operations only using elements derived from their h_{sid} are unobservable, all other operations are observable). In addition to that, it thwarts the attack from the previous section: when

[15] In Groth16, verification happens to only consists of pairing operations and the group operations on $\mathbb{G}_t$, but this does not meaningfully change the argument. Following Functionality 3, the pairing operations would be observable. Furthermore, it is easy to imagine reasonable protocols that involve group operations on input elements.

Functionality 15: Failed attempt 2 for $\mathcal{G}$-oGG

$\mathcal{G}$-oGG maintains the following state:

- $\tau : \mathbb{Z}_p \to S$ a random encoding function
- $V_{sid} \subseteq S$ initially empty sets of group elements belonging to session *sid*
- *Ob* initially empty list of observable actions

```
GetGen_{sid}()
1: if $h_{sid} = \perp$ then
2:     $h_{sid} \xleftarrow{\$} S$
3:     $V_{sid} \leftarrow \{h_{sid}\}$
4: return $h_{sid}$

Observe_{sid}()
5: return $Ob$

Op_{sid}(g_1, g_2, a_1, a_2)
6: assert $(g_1, g_2, a_1, a_2) \in S^2 \times \mathbb{Z}_p^2$
7: $h \leftarrow \tau(a_1\tau^{-1}(g_1) + a_2\tau^{-1}(g_2))$
8: for all sessions $sid'$ do // incl. $sid' = sid$
9:     if $g_1, g_2 \in V_{sid'}$ then
10:        $V_{sid'} \leftarrow V_{sid'} \cup \{h\}$
11: if $h \notin V_{sid}$ then
12:     $Ob \leftarrow Ob : [(\text{Op}, g_1, g_2, a_1, a_2, h)]$
13: return $h$
```

the environment computes $g_{\text{intermediate}}$ in session sid', it will also be added to V_{sid}. However, the environment can sidestep this mechanism quite easily: instead of computing $g_{\text{intermediate}}$ as $\sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid}$, it computes it as $\underline{h_{sid'}} + \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid} - \underline{h_{sid'}}$. The resulting group element is the same, but because the first term $h_{sid'}$ is not in V_{sid}, none of the operations/intermediate results are added to V_{sid}. As a result, with a minimal change to the attack above, the environment can still distinguish the real world from the ideal world, using unintended leakage exposed by Functionality 15.

This sort of sidestepping motivates the final observation rule of Functionality 13. In our final observation rule, the polynomial corresponding to the element $g_{\text{intermediate}}$ computed as $h_{sid'} + \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid} - h_{sid'}$ is the same as the polynomial when computing the same element as $\sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid}$. In both cases, the polynomial corresponding to $g_{\text{intermediate}}$ is the same, namely $\mathsf{X}_{sid'} + \sum_{i=0}^{m} a_i u_i(x) \cdot \mathsf{X}_{sid} - \mathsf{X}_{sid'} = \sum_{i=0}^{m} a_i u_i(x) \cdot \mathsf{X}_{sid}$. As a result, Functionality 13 treats the group element $g_{\text{intermediate}}$ as part of session *sid* and exhibits observability behavior accordingly.

Third attempt: More aggressively adding elements to sessions. The issue with the previous Functionality 15 can be seen as a failure to keep track of group elements. When the environment adds some other sessions' generator $h_{sid'}$ to some $g \in V_{sid}$, the functionality loses track that the resulting group element has anything to do with session *sid*. One attempt to fix this issue is to keep track of session associations more aggressively, i.e. instead of saying that $g_1 + g_2 \in V_{sid}$ if both g_1, g_2 are in V_{sid}, we say that $g_1 + g_2 \in V_{sid}$ if g_1 *or* g_2 is in V_{sid}. As a result, an element $h_{sid'} + h_{sid}$ belongs to *both* $V_{sid'}$ and V_{sid} instead of neither. This is formalized in Functionality 16.

This approach indeed solves all issues with the environment checking whether $g_{\text{intermediate}}$ is in V_{sid}. Whenever the environment computes $g_{\text{intermediate}}$ (in any way), or any element that is losely associated with h_{sid}, it will be added to V_{sid}.

This rule, however, is *too* eager to add elements to V_{sid}, which allows the environment to evade observation. More concretely, say, the environment wants to compute $5 \cdot h_{sid}$ in session sid'. Intuitively, this operation must be observable. However, the environment can escape observability by first computing $h_{sid'} + h_{sid} - h_{sid'}$, which inappropriately adds h_{sid} to $V_{sid'}$. Afterwards, the computation of $5 \cdot h_{sid}$ in session sid' is unobservable because $h_{sid} \in V_{sid'}$. In other words, this rule allows the environment to effectively add foreign elements to its own session, and evade observability. In our final observation rule (Functionality 13), this is again not an issue because the terms $\pm h_{sid'}$ cancel out in the polynomial representations.

Functionality 16: Failed attempt 3 for $\mathcal{G}$-oGG

$\mathcal{G}$-oGG maintains the following state:

- $\tau : \mathbb{Z}_p \to S$ a random encoding function
- $V_{sid} \subseteq S$ initially empty sets of group elements belonging to session sid
- Ob initially empty list of observable actions

```
GetGen_{sid}()
1: if $h_{sid} = \perp$ then
2:     $h_{sid} \xleftarrow{\$} S$
3:     $V_{sid} \leftarrow \{h_{sid}\}$
4: return $h_{sid}$

Observe_{sid}()
5: return $Ob$

Op_{sid}(g_1, g_2, a_1, a_2)
6: assert $(g_1, g_2, a_1, a_2) \in S^2 \times \mathbb{Z}_p^2$
7: $h \leftarrow \tau(a_1\tau^{-1}(g_1) + a_2\tau^{-1}(g_2))$
8: for all sessions $sid'$ do // incl. $sid' = sid$
9:     if $g_1 \in V_{sid'} \vee g_2 \in V_{sid'}$ then
10:        $V_{sid'} \leftarrow V_{sid'} \cup \{h\}$
11: if $h \notin V_{sid}$ then
12:     $Ob \leftarrow Ob : [(\mathrm{Op}, g_1, g_2, a_1, a_2, h)]$
13: return $h$
```

Solution: The polynomial-based observation rule. Reflecting on the failed attempts, we can distill three requirements for a good $\mathcal{G}$-oGG observation rule. From the first attempt (Functionality 14), we have learned that the fact whether or not an element is observable must not depend on whether the element has been computed by honest parties or by the environment.

The second and third attempts (Functionalities 15 and 16) essentially suffer from issues related to mixing generators. The following requirements are derived from these attempts:

- The element $h_{sid'} + 5 \cdot h_{sid} - h_{sid'}$ must be added to V_{sid}.
  - This is needed to ensure the environment cannot check whether $5 \cdot h_{sid}$ has been computed (added to V_{sid}) by honest parties before.
- The element $h_{sid'} + 5 \cdot h_{sid}$ must not be added to $V_{sid'}$.
  - This is needed so that the environment cannot evade observability by simply adding $h_{sid'}$, doing some unobservable computation in session sid', and removing $h_{sid'}$ again at the end.

When using a simple infection-based mechanism as in Functionalities 15 and 16, both of these points cannot be true at the same time. Such a mechanism either assigns $h_{sid'} + 5 \cdot h_{sid}$ and $h_{sid'} + 5 \cdot h_{sid} - h_{sid'}$ to both $V_{sid}, V_{sid'}$ (Functionality 16) or to neither (Functionality 15).

With the polynomial-based observation rule of Functionality 13, we can satisfy both requirements, by keeping track of the makeup of an element itself. The polynomial-based rule can look at $h_{sid'} + 5 \cdot h_{sid}$ as the polynomial $\mathsf{X}_{sid'} + 5\mathsf{X}_{sid}$ and conceptually assign it to neither V_{sid} nor $V_{sid'}$ (as it is a mix of multiple generators). It looks at $h_{sid'} + 5 \cdot h_{sid} - h_{sid'}$ as the polynomial $5\mathsf{X}_{sid}$ and conceptually assigns it to V_{sid} (as it is not a mix of multiple generators).

E FindRep compatible with $\mathbb{G}_\mathrm{t}$

For the sake of simplicity, the GetRep oracle and FindRep algorithm are only presented for $\mathbb{G}_1$ and $\mathbb{G}_2$ elements in the main body. In Function 2, we show FindRep for $\mathbb{G}_\mathrm{t}$ elements. In Oracle 1, we show the corresponding GetRep oracle for $\mathbb{G}_\mathrm{t}$ elements.

To accommodate pairing operations, both GetRep$(\mathrm{t}, \cdots)$ as well as FindRep$(\mathrm{t}, \cdots)$ take bases $B^{(1)}, B^{(2)}, B^{(\mathrm{t})}$ for all three groups as input. The output of FindRep is adapted so that it can express pairing operations of basis elements. For example, if g_1, g_2, g_t are basis

Function 2: FindRep for $\mathbb{G}_t$

```
$\text{FindRep}(t, h^*, Ob_{sid}, B^{(1)}, B^{(2)}, B^{(t)})$
Parse $B^{(i)} = (B^{(i)}_1, \dots, B^{(i)}_{n_i})$
$Rep_1[h] \leftarrow 0 \in \mathbb{Z}_p^{n_1}$ initially for all $h \in S_1$
for $b \in [n_1]$ do $Rep_1[B^{(1)}_b] \leftarrow (\text{Kronecker}_{k,b})_{k=1}^{n_1}$
$Rep_2[h] \leftarrow 0 \in \mathbb{Z}_p^{n_2}$ initially for all $h \in S_2$
for $b \in [n_2]$ do $Rep_2[B^{(2)}_b] \leftarrow (\text{Kronecker}_{\ell,b})_{\ell=1}^{n_2}$
$Rep_t[h] \leftarrow 0 \in \mathbb{Z}_p^{n_t + n_1 \cdot n_2}$ initially for all $h \in S_t$
for $b \in [n_t]$ do $Rep_t[B^{(t)}_b] \leftarrow (\text{Kronecker}_{j,b})_{j=1}^{n_t + n_1 \cdot n_2}$
// We write $Rep_t[h] = (a_j)_{j\in[n_t]}, (a_{k,\ell})_{k\in[n_1],\ell\in[n_2]})$, where $a_{k,\ell}$ are the coefficients for the pairings of baseis
  elements $B^{(1)}_k$ and $B^{(2)}_\ell$.
for $ob \in Ob_{sid}$ do // In the order entries appear in $Ob_{sid}$
    if $ob = (\text{Op}, i, g_1, g_2, a_1, a_2, h) \wedge i \in \{1, 2, t\}$ then // Group operation
        $Rep_i[h] \leftarrow a_1 \cdot Rep_i[g_1] + a_2 \cdot Rep_i[g_2]$
    if $ob = (\text{Pair}, t, g_1, g_2, h)$ then // Pairing operation
        $Rep_t[h] \leftarrow (0^{n_t}, Rep_1[g_1] \otimes Rep_2[g_2])$ // $c \otimes d := (c_k \cdot d_\ell)_{k\in[n_1],\ell\in[n_2]}$
return $Rep_t[h^*]$
```

Oracle 1: $\mathcal{G}$-oSG extension: GetRep for $\mathbb{G}_t$

```
$\text{GetRep}_{sid}(t, h^*, B^{(1)}, B^{(2)}, B^{(t)})$
assert $h^* \in \text{im}(\tau_t)$ and $B^{(i)} \subseteq C_i$ for $i \in \{1, 2, t\}$
Parse $B^{(i)} = (B^{(i)}_1, \dots, B^{(i)}_{n_i})$
$((a_j)_{j\in[n_t]}, (a_{k,\ell})_{k\in[n_1],\ell\in[n_2]}) \leftarrow \text{FindRep}(h^*, Ob, Ob_{sid}, B^{(1)}, B^{(2)}, B^{(t)})$
assert $\sum_{j=1}^{|B^t|} a_j \cdot \tau_t^{-1}(B^{(t)}_j) + \sum_{k=1}^{|B^{(1)}|} \sum_{\ell=1}^{|B^{(2)}|} a_{k,\ell} \cdot \tau_1^{-1}(B^{(1)}_k) \cdot \tau_2^{-1}(B^{(2)}_\ell) = \tau_t^{-1}(h^*) \mod \langle \mathsf{Var}_{-sid}, \tau_1^{-1}(C_1 \setminus$
  $B^{(1)}), \tau_2^{-1}(C_2 \setminus B^{(2)}), \tau_t^{-1}(C_t \setminus B^{(t)})\rangle_{\mathbb{Z}_p[\mathsf{Var},\mathsf{SimVar}]}$
return $((a_j)_{j\in[n_t]}, (a_{k,\ell})_{k\in[n_1],\ell\in[n_2]})$
```

elements, we want to be able to output coefficients $a_1, a_{1,1}$ such that $h^* = a_1 \cdot g_t + a_{1,1} \cdot e(g_1, g_2)$ (modulo some terms).

Group operations are handled in FindRep as usual, adding up the representation vectors *Rep* of the operands. Pairing operations are handled in the natural way, too, pairwise multiplying the known representation vector entries (mirroring $e(\sum_k y_k, \sum_\ell z_\ell) = \sum_k \sum_\ell e(y_k, z_\ell)$).

Similar to the proof of Lemma 5 (see Appendix I), we can also show that whenever FindRep sets some $Rep_i[h]$ value, then that value is a good representation of the element $\tau^{-1}(h)$ modulo $\langle \mathsf{Var}_{-sid}, \tau_1^{-1}(C_1 \setminus B^{(1)}), \tau_2^{-1}(C_2 \setminus B^{(2)}), \tau_t^{-1}(C_t \setminus B^{(t)})\rangle_{\mathbb{Z}_p[\mathsf{Var},\mathsf{SimVar}]}$. This means that there is no correctness guarantee for terms that involve (1) variables from other sessions (Var_{-sid}) or (2) outputs of symbolic computations not passed as input ($\tau_i^{-1}(C_i \setminus B^{(i)})$, which are, in a sense, "missing" basis elements). However, all other terms get correct coefficients from FindRep, where "correct" means consistent to $\tau_t(h)$. In particular, if some $\mathbb{G}_t$ element can be written without involvement of foreign session variables and one passes all output of ComputeSymbolic as basis elements into FindRep, then the output encodes h^* exactly. These guarantees are encoded in Oracle 1.

F Proof of Lemma 3

First, note that $\mathcal{G}$-oGG samples all group element encodings in the beginning ($\tau_i \xleftarrow{\$} \text{Inj}(\mathbb{Z}_p, S_i)$). In contrast, $\mathcal{G}$-oSG *lazily* samples the values for τ_i (this is because there *are no* injective functions from $\mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$ to S_i). To make $\mathcal{G}$-oGG more like $\mathcal{G}$-oSG in that regard, consider

Functionality 17: $\mathcal{G}$-oGG with lazily sampled τ_i

Differences to original $\mathcal{G}$-oGG (Functionality 3) are marked with purple.

```
Init()  // Invoked only upon creation
 1: for $i \in \{1,2,\mathrm{t}\}$ do
 2:     $\tau_i \leftarrow \{\}$
 3:     $g_i \leftarrow \textsc{Tau}_{sid}(i,1)$
 4:     $\mathsf{R}_i[g_i] \leftarrow \{1\}$

CanonicalGen_sid(i)
 5: return $\tau_i(1)$

Observe_sid()
 6: return $Ob$

Tau_sid(i, a)  // internal
 7: if $\tau_i(a) = \perp$ then
 8:     $\tau_i(a) \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau_i)$
 9: return $\tau_i(a)$

Op_sid(i, g_1, g_2, a_1, a_2)
10: assert $(g_1, g_2, a_1, a_2) \in S_i^2 \times \mathbb{Z}_p^2$
11: for $j \in \{1,2\}$ do
12:     $\textsc{Touch}_{sid}(i, g_j)$
13: $h \leftarrow \textsc{Tau}_{sid}(i, a_1\tau_i^{-1}(g_1) + a_2\tau_i^{-1}(g_2))$
14: $\mathsf{R}_i[h] \leftarrow \mathsf{R}_i[h] \cup (a_1\mathsf{R}_i[g_1] + a_2\mathsf{R}_i[g_2])$
15: if $\exists \mathsf{f} \in \mathsf{R}_i[h] : f \notin \mathsf{Legal}_{sid}$ then
16:     $Ob \leftarrow Ob : [(\textsc{Op}, i, g_1, g_2, a_1, a_2, h)]$
17: return $h$

Touch_sid(i, g)
18: if $\mathsf{R}_i[g] = \varnothing$ then
19:     Initialize fresh variable $\mathsf{X}$
20:     $\mathsf{Var}_{i,sid} \leftarrow \mathsf{Var}_{i,sid} : [\mathsf{X}]$
21:     $\mathsf{R}_i[g] \leftarrow \{\mathsf{X}\}$
22:     $x \xleftarrow{\$} \mathbb{Z}_p \setminus \mathrm{dom}(\tau_i)$
23:     $\tau_i(x) \leftarrow g$

Pair_sid(g_1, g_2)
24: assert $(g_1, g_2) \in S_1 \times S_2$
25: for $i \in \{1,2\}$ do
26:     $\textsc{Touch}_{sid}(i, g_i)$
27: $h \leftarrow \textsc{Tau}_{sid}(\mathrm{t}, \tau_1^{-1}(g_1) \cdot \tau_2^{-1}(g_2))$
28: $\mathsf{R}_\mathrm{t}[h] \leftarrow \mathsf{R}_\mathrm{t}[h] \cup (\mathsf{R}_1[g_1] \cdot \mathsf{R}_2[g_2])$
29: if $\exists \mathsf{f} \in \mathsf{R}_\mathrm{t}[h] : \mathsf{f} \notin \mathsf{Legal}_{sid}$ then
30:     $Ob \leftarrow Ob : [(\textsc{Pair}, \mathrm{t}, g_1, g_2, h)]$
31: return $h$
```

the lazy sampling version of $\mathcal{G}$-oGG in Functionality 17: Whenever $\tau_i(x)$ is first accessed ($x \notin \mathrm{dom}(\tau_i)$), a random unused image $h \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau_i)$ is chosen for x. Whenever $\tau_i^{-1}(h)$ is first accessed ($h \notin \mathrm{im}(\tau_i)$), a random unused preimage $x \xleftarrow{\$} \mathbb{Z}_p \setminus \mathrm{dom}(\tau_i)$ is chosen for h. We push the code for first-access of τ_i into a new internal interface $\textsc{Tau}_{sid}$. We push the code for first-access of τ_i^{-1} into $\textsc{Touch}_{sid}$ because it precedes all accesses of τ_i^{-1}. Of course, this version is perfectly indistinguishable from the original $\mathcal{G}$-oGG (this is easy to see because the $\tau_i : \mathbb{Z}_p \to S_i$ are bijections).

For the proof of Lemma 3, consider the following "hybrid" functionality $\mathcal{G}$-oHG–b, which outwardly behaves like $\mathcal{G}$-oSG for $b = 0$ and like (the lazy sampling version of) $\mathcal{G}$-oGG for $b = 1$. This can be checked by inspection.

Note that the lazy sampling of τ_i' preimages x' in Touch first tries to use some uniform $x \xleftarrow{\$} \mathbb{Z}_p$ in Line 31, but falls back to $x' \xleftarrow{\$} \mathbb{Z}_p \setminus \mathrm{dom}(\tau_i')$ (Line 33) in case there is a collision. This way of lazily sampling preimages is perfectly equivalent to the lazy sampling in Functionality 17. However, the intermediate uniform preimage x will allow us to apply Schwartz-Zippel to some meaningful uniform and independent x values.

The proof will establish that there is no difference between $\mathcal{G}$-oHG–0 and $\mathcal{G}$-oHG–1 unless during execution of $\mathcal{G}$-oHG–0, we run into two polynomials $\mathsf{f} \neq \mathsf{f}' \in \mathsf{P}$ that collide, i.e. $\mathsf{f}(Val) = \mathsf{f}'(Val)$. Schwartz-Zippel (Lemma 1) establishes that such collisions are unlikely.

Proof (Lemma 3). Consider the event "coll" that at some point during execution of $\mathcal{G}$-oHG–0, there are $\mathsf{f}, \mathsf{f}' \in \mathsf{P}$ such that $\mathsf{f} \neq \mathsf{f}'$ but $\mathsf{f}(Val) = \mathsf{f}'(Val)$. Because polynomials in P are at most of degree 2, the values in Val are chosen uniformly and independently at random, and the set P of polynomials does not depend on Val (all responses of $\mathcal{G}$-oHG–0 to $\mathcal{B}$ are independent

Functionality 18: $\mathcal{G}$-oHG–b

- $\tau_i : \mathbb{Z}_p[\mathsf{Var}] \to S_i$ // as in $\mathcal{G}$-oSG
- $\tau'_i : \mathbb{Z}_p \to S_i$ // as in $\mathcal{G}$-oGG
- P set of polynomials seen during the game // bookkeeping for proof
- R_i initially empty map from representations to sets of polynomials // as in $\mathcal{G}$-oGG
- $\mathsf{Var}_{i,sid}$ initially empty lists of polynomial variables // as in $\mathcal{G}$-oSG
- $Val_{i,sid}$ initially empty lists of random values corresponding to the variables of $\mathsf{Var}_{i,sid}$ // Uniform version (with potential collisions) of Val'. Used for Schwartz-Zippel
- $Val'_{i,sid}$ initially empty lists of random values corresponding to the variables of $\mathsf{Var}_{i,sid}$ // Version that potentially deviates from Val because Val' does not contain duplicates. Used to bridge $\mathcal{G}$-oSG into $\mathcal{G}$-oGG in the proof, where polynomials f in $\mathcal{G}$-oSG correspond to scalars $\mathsf{f}(Val')$ in $\mathcal{G}$-oGG
- Ob initially empty list of (globally) observable actions // as in $\mathcal{G}$-oGG, $\mathcal{G}$-oSG

We write

- We write $\mathsf{Var}_{sid}, \mathsf{Var}$ as before. We analogously define Val_{sid} and Val.
- $\mathsf{Legal}_{sid} = \langle \mathsf{Var}_{sid} \rangle_{\mathbb{Z}_p[\mathsf{Var}_{sid}]}$ // As in $\mathcal{G}$-oGG and $\mathcal{G}$-oSG (the formulation in the latter is equivalent since $\mathsf{SimVar}_{sid} = ()$ is empty in the context of Lemma 3)

```
INIT() // Invoked only upon creation
for $i \in \{1,2,\mathrm{t}\}$ do
    $\tau_i \leftarrow \{\}$
    $\tau'_i \leftarrow \{\}$
    $g_i \leftarrow$ TAU$(i,1,1)$
    $\mathsf{R}[g_i] \leftarrow \{1\}$

CANONICALGEN$_{sid}(i)$
return TAU$(i,1,1)$

TAU$(i,\mathsf{f},a)$ // internal
$\mathsf{P} \leftarrow \mathsf{P} \cup \{\mathsf{f}\}$
if $b = 0 \wedge \tau_i(\mathsf{f}) = \perp \vee b = 1 \wedge \tau'_i(a) = \perp$ then
    if $b = 0$ then $h \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau_i)$
    if $b = 1$ then $h \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau'_i)$
    $\tau_i(\mathsf{f}) \leftarrow h$
    $\tau'_i(a) \leftarrow h$
if $b = 0$ then return $\tau_i(\mathsf{f})$
if $b = 1$ then return $\tau'_i(a)$

OP$_{sid}(i,g_1,g_2,a_1,a_2)$
assert $(g_1,g_2,a_1,a_2) \in S_i^2 \times \mathbb{Z}_p^2$
for $j \in \{1,2\}$ do
    TOUCH$_{sid}(i,g_j)$
$\mathsf{f} \leftarrow a_1\tau_i^{-1}(g_1) + a_2\tau_i^{-1}(g_2)$
$a \leftarrow a_1\tau_i'^{-1}(g_1) + a_2\tau_i'^{-1}(g_2)$
$h \leftarrow$ TAU$(i,\mathsf{f},a)$
$\mathsf{R}_i[h] \leftarrow \mathsf{R}_i[h] \cup (a_1\mathsf{R}_i[g_1] + a_2\mathsf{R}_i[g_2])$
if $b = 0 \wedge \mathsf{f} \notin \mathsf{Legal}_{sid}$ then
    $Ob \leftarrow Ob : [($OP$, i, g_1, g_2, a_1, a_2, h)]$
if $b = 1 \wedge \mathsf{R}_i[h] \not\subseteq \mathsf{Legal}_{sid}$ then
    $Ob \leftarrow Ob : [($OP$, i, g_1, g_2, a_1, a_2, h)]$
return $h$

TOUCH$_{sid}(i,g)$
if $b = 0 \wedge g \notin \mathrm{im}(\tau_i) \vee b = 1 \wedge \mathsf{R}_i[g] = \varnothing$ then
    Initialize a fresh variable $\mathsf{X}$
    $\mathsf{Var}_{i,sid} \leftarrow \mathsf{Var}_{i,sid} : [\mathsf{X}]$
    $\tau_i(\mathsf{X}) \leftarrow g$
    $\mathsf{R}_i[g] \leftarrow \{\mathsf{X}\}$
    $x \xleftarrow{\$} \mathbb{Z}_p$
    if $\tau'_i(x) \neq \perp$ then
        $x' \xleftarrow{\$} \mathbb{Z}_p \setminus \mathrm{dom}(\tau'_i)$
    else
        $x' \leftarrow x$
    $Val_{i,sid} \leftarrow Val_{i,sid} : [x]$
    $Val'_{i,sid} \leftarrow Val'_{i,sid} : [x']$
    $\tau'_i(x') \leftarrow g$
    $\mathsf{P} \leftarrow \mathsf{P} \cup \{\mathsf{X}\}$

OBSERVE$_{sid}()$
return $Ob$

PAIR$_{sid}(g_1,g_2)$
assert $(g_1,g_2) \in S_1 \times S_2$
for $i \in \{1,2\}$ do
    TOUCH$_{sid}(i,g_i)$
$\mathsf{f} \leftarrow \tau_1^{-1}(g_1) \cdot \tau_2^{-1}(g_2)$
$a \leftarrow \tau_1'^{-1}(g_1) \cdot \tau_2'^{-1}(g_2)$
$h \leftarrow$ TAU$(\mathrm{t},\mathsf{f},a)$
$\mathsf{R}_{\mathrm{t}}[h] \leftarrow \mathsf{R}_{\mathrm{t}}[h] \cup (\mathsf{R}_1[g_1] \cdot \mathsf{R}_2[g_2])$
if $b = 0 \wedge \mathsf{f} \notin \mathsf{Legal}_{sid}$ then
    $Ob \leftarrow Ob : [($PAIR$, \mathrm{t}, g_1, g_2, h)]$
if $b = 1 \wedge \mathsf{R}_{\mathrm{t}}[h] \not\subseteq \mathsf{Legal}_{sid}$ then
    $Ob \leftarrow Ob : [($PAIR$, \mathrm{t}, g_1, g_2, h)]$
return $h$
```

of Val), we can apply Lemma 1 pairwise to all f, f'. This gives us that $\Pr[\mathsf{f}(\mathit{Val}) - \mathsf{f}'(\mathit{Val}) = 0] \leq \deg(\mathsf{f} - \mathsf{f}')/p \leq 2/p$ for all $\{\mathsf{f}, \mathsf{f}'\} \in \binom{\mathsf{P}}{2}$. Every query to $\mathcal{B}$'s oracles adds at most three new polynomials to P, in addition to the initial entry $1 \in \mathsf{P}$. With union bound over all $\binom{|\mathsf{P}|}{2} \leq \binom{3q+1}{2}$ pairs, we get

$$\Pr[\mathrm{coll}] = \Pr[\exists \mathsf{f} \neq \mathsf{f}' \in \mathsf{P}: \ \mathsf{f}(\mathit{Val}) = \mathsf{f}'(\mathit{Val})] \leq \binom{3q+1}{2} \cdot 2/p.$$

It remains to show that $\left|\Pr\left[\mathcal{B}^{\mathcal{O}_{\mathrm{real}}} = 1\right] - \Pr\left[\mathcal{B}^{\mathcal{O}_{\mathrm{symb}}} = 1\right]\right| \leq \Pr[\mathrm{coll}]$. For this, we claim that if coll does not occur, then $\mathcal{G}$-`oHG–0` and $\mathcal{G}$-`oHG–1` behave exactly the same. This implies the bound above via difference lemma.

To check this claim, we consider two invariants that hold before and after any $\mathcal{G}$-`oHG–0` oracle query, assuming that $\neg$coll.

Invariant 1. First, we characterize τ_i, τ'_i, and R_i by putting them into one common table. We define tables $T_i : S_i \to S_i \times \mathbb{Z}_p[\mathsf{Var}] \times 2^{\mathbb{Z}_p} \times 2^{\mathbb{Z}_p[\mathsf{Var}]}$ with

$$T_i(h) = (h, \tau_i^{-1}(h), \tau_i'^{-1}(h), \mathsf{R}_i[h]).$$

Note that τ_i is injective by design of $\mathcal{G}$-`oHG–0`, so τ_i^{-1} is either $\perp$ or some unique single value. τ'_i is not necessarily injective in $\mathcal{G}$-`oHG–0`, so $\tau_i'^{-1}(h)$ returns the set of all preimages of h. The first invariant is: Before and after any invocation of the oracles, it holds that for all $i \in \{1, 2, \mathrm{t}\}, h \in S_i$, we have

$$\begin{aligned} & T_i(h) = (h, \perp, \varnothing, \varnothing) \\ \text{or } & T_i(h) = (h, \mathsf{f}, \{\mathsf{f}(\mathit{Val}')\}, \{\mathsf{f}\}) \text{ for some } \mathsf{f} \in \mathsf{P} \end{aligned}$$

In addition, for all input (i, f, a) ever supplied to TAU, it holds that $\mathsf{f}(\mathit{Val}') = a$.

We call the first kind of entry in T_i "empty" and the second kind "non-empty". Intuitively, this invariant establishes a strong connection between the symbolic polynomials of τ_i, the concrete discrete logarithms of τ'_i, and the $\mathcal{G}$-`oGG` bookkeeping polynomials R_i. Namely, $\mathsf{R}_i[h]$ is essentially just $\tau_i^{-1}(h)$, and we get from $\tau_i^{-1}(h) \in \mathbb{Z}_p[\mathsf{Var}]$ to $\tau_i'^{-1}(h) \in \mathbb{Z}_p$ by just plugging in the concrete discrete logarithms Val'.

Invariant 2. The second invariant is: Before and after any invocation of the oracles, it holds that $\mathit{Val} = \mathit{Val}'$. This invariant claims, essentially, that there is no disparity between the uniformly chosen preimages $\mathit{Val} \overset{\$}{\leftarrow} \mathbb{Z}_p$ and the (collisionless) preimages Val' used for τ'_i.

Corollaries of invariants 1,2. As a consequence of the first invariant, we immediately get that $\mathrm{im}(\tau_i) = \mathrm{im}(\tau'_i)$ (given that any h is either in the image of none of the τ_i, τ'_i functions, or in both of them). Furthermore, because τ_i, τ'_i are functions, any $\mathsf{f} \in \mathbb{Z}_p$ appears in T_i at most once and any $a \in \mathbb{Z}_p$ appears in T_i at most once.

Claim 1 (Invariants hold). *Assuming* $\neg$coll*, invariant 1 and invariant 2 both hold before and after any oracle call to* $\mathcal{G}$-`oHG–0`*.*

Proof: **After Init.** The first invariant is clearly fulfilled in the beginning, after INIT, where the T_i contain the canonical generator entries $(g_i, 1, \{1\}, \{1\})$, and the entries for all other $h \neq g_i$ are of the empty kind. The second invariant is trivially fulfilled in the beginning, given that $\mathit{Val} = \mathit{Val}' = ()$ is empty.

In the following, we assume that the invariants hold before any oracle call, and prove that they are preserved, assuming $\neg$coll.

Preservation of invariant 1. The first invariant is preserved through TOUCH$_{sid}(i, g)$, which either does nothing (if $g \in \mathrm{im}(\tau_i)$, i.e. if the T entry for g is non-empty), or otherwise adds the non-empty entry $T(g) = (g, \mathsf{X}, \{x'\}, \{\mathsf{X}\})$ where $x' = \mathsf{X}(\mathit{Val}')$ by design. TOUCH never overwrites any existing τ_i or τ_i' values, meaning that no other entry of T changes.

The invariant is also preserved by queries to $\text{OP}_{\mathit{sid}}(i, g_1, g_2, a_1, a_2)$ and $\text{PAIR}_{\mathit{sid}}(g_1, g_2)$: For both interfaces, the invariant holds after the internal call to TOUCH (Line 17 and 43). In particular, because of invariant 1's guarantees for g_1, g_2, we get that f and a, as computed in Line 18 and 19 and Line 44 and 45 also fulfill $\mathsf{f}(\mathit{Val}') = a$. Hence $\text{TAU}_{\mathit{sid}}(i, \mathsf{f}, a)$ is called with $a = \mathsf{f}(\mathit{Val}')$ in Line 20 and 46, which fulfills the "for all input (i, f, a) ever supplied to TAU, it holds that $\mathsf{f}(\mathit{Val}') = a$" part of invariant 1.

There are now two cases. (1) if $T(h)$ is non-empty, TAU changes nothing, preserving the invariant. (2) if $T(h)$ is empty, then $\text{TAU}_{\mathit{sid}}(i, \mathsf{f}, \mathsf{f}(\mathit{Val}'))$ sets the previously empty entry $T(h)$ to $(h, \mathsf{f}, \{\mathsf{f}(\mathit{Val}')\}, \cdot)$.

For case (2), note that it might happen that the assignment $\tau_i'(a) \leftarrow h$ in Line 12 overwrites a prior value $\tau_i'(a) = h_2$, which would violate the invariant for $T(h_2)$. However, assume there were some non-empty entry $(h_2, \mathsf{f}_2, \{a\}, \cdot)$ containing a before executing TAU. Then $a = \mathsf{f}_2(\mathit{Val}')$ by invariant 1. Hence $\mathsf{f}(\mathit{Val}') = \mathsf{f}'(\mathit{Val}') = a$. By invariant 2, $\mathsf{f}(\mathit{Val}) = \mathsf{f}'(\mathit{Val})$, and by injectivity of τ_i', $\mathsf{f} \neq \mathsf{f}_2$, i.e. we get a collision between $\mathsf{f}(\mathit{Val})$ and $\mathsf{f}_2(\mathit{Val})$. Overall, Line 12 does not overwrite any prior $\tau_i'(a) = h_2$ value unless the event coll happens.

Afte the internal call to TAU, because $\mathsf{R}_i[g_\ell] = \{\tau_i^{-1}(g_\ell)\}$ for the inputs g_1, g_2 by invariant 1, OP/PAIR sets $\mathsf{R}[h]$ to $\{\mathsf{f}\}$ in Line 21 and 47, which gives us the entry $T_i(h)$ of the form $(h, \mathsf{f}, \{\mathsf{f}(\mathit{Val}')\}, \{\mathsf{f}\})$, preserving invariant 1, as required.

Preservation of invariant 2. The second invariant is preserved by TOUCH assuming $\neg$coll: The only way for Val' to deviate from Val is if the condition $\tau_i'(x) \neq \bot$ in Line 32 becomes true. In the following, we argue that this can only happen if the event coll occurs. Assume, that the condition in line Line 32 is true. That implies that there is a $T(h) = (h, \mathsf{f}, \{x\}, \cdot)$ entry containing x. By invariant 1, $T(h)$ is well-formed, i.e. $\mathsf{f}(\mathit{Val}') = x = \mathsf{X}(\mathit{Val}', x)$. Furthermore, because invariant 2 holds before TOUCH, we have $\mathit{Val}' = \mathit{Val}$, hence $\mathsf{f}(\mathit{Val}') = \mathsf{f}(\mathit{Val}) = \mathsf{f}(\mathit{Val}, x)$ (the last equality holds because f does not contain X). In this scenario, TOUCH would proceed to compute the new $\mathit{Val} \leftarrow \mathit{Val} : [x]$, and X is added to P. This means that after TOUCH, there is a collision between $\mathsf{f} \neq \mathsf{X}$ (inequality because X is a fresh variable) with $\mathsf{f}(\mathit{Val}) = \mathsf{X}(\mathit{Val}) = x$, which violates the assumption that $\neg$coll. Overall, this shows that whenever the condition in Line 32 becomes true, this induces the event coll (which we assume does not happen).

The second invariant is trivially preserved by all other oracle queries (using the argument above for when TOUCH is called internally). ■

Having established those two invariants, we can now argue that as long as the event coll does not occur, $\mathcal{G}$-oHG–0 behaves exactly like $\mathcal{G}$-oHG–1.

Claim 2 (Equivalence until $\neg$coll). *Assuming $\neg$coll, the behavior of $\mathcal{G}$-oHG–0 is exactly like that of $\mathcal{G}$-oHG–1.*

Proof: We go through the lines in which $\mathcal{G}$-oHG–b reads b and argue that the concrete value of b makes no difference, assuming $\neg$coll.

– Regarding Line 8: no difference. By invariant 1, TAU is called with input of the form $(i, \mathsf{f}, a = \mathsf{f}(\mathit{Val}'))$. Consider two cases: (1) if $\tau_i(\mathsf{f}) \neq \bot$, then by invariant 1, there is an entry $T_i(h) = (h, \mathsf{f}, \{a\}, \cdot)$, hence $\tau_i'(a) \neq \bot$. (2) If $\tau_i'(a) \neq \bot$, then also $\tau_i(\mathsf{f}) \neq \bot$: In this case, there is already an entry $T_i(h_2) = (h_2, \mathsf{f}_2, \{a\}, \cdot)$ containing a. Assume, for contradiction, that $\tau_i(\mathsf{f}) = \bot$. Then TAU would reassign $\tau_i'(a)$ to $h \neq h_2$, making the entry $T_i(h_2) = (h_2, \mathsf{f}_2, \underline{\varnothing}, \cdot)$ violate invariant 1 after the update. The invariant cannot be violated, hence necessarily, $\text{im}(\mathsf{f}') \neq \bot$. Overall, either both $\tau_i(\mathsf{f})$ and $\tau_i'(a)$ are $\bot$, or neither.

Functionality 19: Interfaces modeling either $\mathcal{O}$, ComputeConcrete ($b = 0$) or $\mathcal{O}$, ComputeAtomic ($b = 1$) for Lemma 4

Differences with $\mathcal{O}$, ComputeConcrete, ComputeAtomic are highlighted in purple.

- $Q_i \subseteq S_i$ initially empty set containing what would be the intermediate results of ComputeConcrete$(i, \dots)$

```
Compute_sid(i, (h_j, f_j)_{j=1}^n)
1: if $\exists j : h_j \in Q_i$ then
2:     abort
3: assert $\tau_i^{-1}(h_j) \in \mathsf{Legal}_{sid}$, and $\mathsf{f}_j \in \mathbb{Z}_p[\mathsf{SimVar}_{sid}^{\pm 1}]$ for all $j \in [n]$.
4: for $j \in [n-1]$ do
5:     $\mathsf{t}_j \leftarrow \sum_{\ell=1}^{j} \tau_i^{-1}(h_\ell) \cdot \mathsf{f}_\ell(\mathit{SimVal}_{sid})$
6:     PhantomTau_sid$(i, \mathsf{t}_j)$ // Query encoding for partial sum
7: $\mathsf{f} \leftarrow \sum_{\ell=1}^{n} \tau_i^{-1}(h_\ell) \cdot \mathsf{f}_\ell(\mathit{SimVal}_{sid})$
8: $h \leftarrow$ Tau_sid$(i, \mathsf{f})$ // Full sum
9: return $h$

Op, Pair, Observe   As in $\mathcal{G}$-oSG.

Touch_sid(i, g)
10: if $g \in Q_i$ then
11:     abort
12: if $g \notin \mathrm{im}(\tau_i)$ then
13:     Initialize a fresh variable $\mathsf{X}$
14:     $\mathsf{Var}_{i,sid} \leftarrow \mathsf{Var}_{i,sid} : [\mathsf{X}]$
15:     $\tau_i(\mathsf{X}) \leftarrow g$

Tau(i, f)
16: if $\tau_i(\mathsf{f}) = \bot$ then
17:     $\tau_i(\mathsf{f}) \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau_i)$
18: $Q_i \leftarrow Q_i \setminus \{\tau_i(\mathsf{f})\}$
19: return $\tau_i(\mathsf{f})$

PhantomTau_sid(i, f)
20: if $\tau_i(\mathsf{f}) = \bot$ then
21:     $\tau_i(\mathsf{f}) \xleftarrow{\$} S_i \setminus \mathrm{im}(\tau_i)$
22:     $Q_i \leftarrow Q_i \cup \{\tau_i(\mathsf{f})\}$
```

- Regarding Line 9 and 10: no difference because $\mathrm{im}(\tau_i) = \mathrm{im}(\tau_i')$ (as observed in the invariant corollaries above)
- Regarding Line 13 and 14: no difference. By invariant 1, Tau is called with input of the form $(i, \mathsf{f}, a = \mathsf{f}(\mathit{Val}'))$. For $b = 0$, using invariant 1, Tau returns the unique h such that $T(h) = (h, \mathsf{f}, a, \cdot)$. For $b = 1$, Tau must return the same h because a only appears once in T (see invariant corollaries).
- Regarding Line 22, 24, 48, 50: no difference because $\mathsf{R}_i[h] = \{\mathsf{f}\}$ by invariant 1.
- Regarding Line 26: no difference because $\mathrm{im}(\tau_i) = \{g \mid \mathsf{R}_i[g] \neq \bot\}$ by inspection.

■

Using the remarks and probability analysis at the beginning of this proof, Claim 2 enables us to apply the difference lemma, concluding the overall proof. □

G Proof of the first part of Lemma 4

G.1 Indistinguishability of ComputeConcrete and ComputeAtomic

The difference between ComputeConcrete and ComputeAtomic is that the former calls Tau on all intermediate results (partial sums) $\mathsf{t}_j \leftarrow \sum_{\ell=1}^{j} \tau_i^{-1}(h_\ell) \cdot \mathsf{f}_\ell(\mathit{SimVal}_{sid})$, whereas the latter only calls Tau on the final result. Functionality 19 encapsulates this essential difference formally: Compute (corresponding to ComputeConcrete / ComputeAtomic) calls a procedure PhantomTau(i, t_j) for the intermediate results t_j, which allocates a random encoding $h_j = \tau_(\mathsf{t}_j)$ if it does not exist already, but also marks that encoding h_j as "intermediate" by adding it to the set Q_i. If the normal Tau(i, t_j) is called (later), h_j ceases to be an intermediate result, and it is removed from Q_i. Hence Q_i represents the set of intermediate results for which $\mathcal{B}$ has not seen the encodings.

Functionality 19 perfectly emulates both $(\mathcal{O}, \textsc{ComputeConcrete})$ and $(\mathcal{O}, \textsc{ComputeAtomic})$, *unless* it aborts.[16] This happens if $\mathcal{B}$ finds one of the intermediate result encodings $h_j \in Q_i$ and queries it to Compute or Touch. At that point, in the ComputeConcrete setting, this encoding exists, meaning that ComputeConcrete does not refuse to work with it, and Touch does nothing. In contrast, in the ComputeAtomic setting, this encoding does not exist, meaning that the assertion in ComputeAtomic fails and Touch would treat the encoding as new, giving it a fresh variable. So in this case, the two settings become distinguishable.

To bound the probability that Functionality 19 aborts, observe the following. Let $Q = Q_1 \cup Q_2 \cup Q_{\mathrm{t}}$.

- There are at most $q' \geq |Q|$ elements in Q because elements are added only for intermediate Compute results.
- $\mathcal{B}$ has no information about the elements of $h \in Q$ except that any such h cannot have been the output of any of the (at most q) Op, Pair queries. As a consequence, guessing an element of Q succeeds with probability at most $|Q|/(p-q) \leq q'/(p-q)$
- $\mathcal{B}$ can make a guess by querying some h to Compute or Touch. Hence $\mathcal{B}$ makes at most $2q + q'$ guesses (note that any of the $\leq q$ queries to Op / Pair causes two Touch queries).
- With union bound, this gives us that $\mathcal{B}$ makes Functionality 19 abort with probability at most $(2q + q') \cdot q'/(p-q)$.

Overall, with the difference lemma, we get that

$$\left| \begin{array}{r} \Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeConcrete}} = 1\right] \\ -\Pr\left[\mathcal{B}^{\mathcal{O},\textsc{ComputeAtomic}} = 1\right] \end{array} \right| \leq \Pr[\text{abort}] \leq (2q + q') \cdot q'/(p-q)$$

as required.

H A technical lemma for switching computation styles in $\mathcal{G}$-oSG

The next lemma (Lemma 7) alleviates a technical concern. Whenever the UC simulator $\mathcal{S}$ makes a group operation query, the result of that query gets assigned a random image in τ_i. As a consequence, in $\mathcal{G}$-oSG, it is no longer true that it does not matter how $\mathcal{S}$ computes a group elements as long as the result is (distributed) the same. We can imagine one simulator using 500 queries to compute a group element, and another simulator using only 100 queries to compute it (e.g., because it uses a trapdoor). Even though both output the same (distribution of) group elements, the environment can potentially distinguish the two by choosing a random $g \xleftarrow{\$} S_i$ and querying it to $\textsc{Touch}_{sid'}(i, g)$. This Touch query fails more often in the presence of the first than the second simulator, because the first one computes many more intermediate results, which increases the chances that the random g is one of them. Of course, S_i is large, and the chances of mounting this distinguishing attack are negligible. The following lemma deals with this issue, showing that, essentially as a corollary of Lemma 4, the way a group element is computed is undetectable. The lemma is used in Claim 6 of the Groth16 proof.

Lemma 7. *Let $\mathcal{A}^{\mathcal{G}\text{-oSG},\mathcal{B}_i}, \mathcal{B}_0^{\textsc{ComputeConcrete}}, \mathcal{B}_1^{\textsc{ComputeConcrete}}$ be algorithms such that both $\mathcal{B}_i$ just output the list of their $h_j \leftarrow \textsc{ComputeConcrete}(i_j, \cdot)$ query results in the format*

[16] We are using here that $\mathcal{B}$ is restricted such that $3q + q' + 1 \leq p$, i.e. we do not run out of unused encoding in either of the settings. Furthermore, by the assertion in Line 49 of ComputeConcrete, none of the Op calls are (globally) observable.

$((i_1, h_1), (i_2, h_2), \dots)$. *We say that* $\mathcal{B}_0, \mathcal{B}_1$ *are* perfectly equivalent *(w.r.t.* $\mathcal{A}$*) if for any invocation* $\mathcal{B}(x)$ *that* $\mathcal{A}$ *makes,* $\Pr[y = (\tau_{i_j}^{-1}(h_j))_{j=1}^n \mid (i_j, h_j)_{j=1}^n \leftarrow \mathcal{B}_0(x)] = \Pr[y = (\tau_{i_j}^{-1}(h_j))_{j=1}^n \mid (i_j, h_j)_{j=1}^n \leftarrow \mathcal{B}_1(x)]$ *for all* y*.*

Let q *be an upper bound on the number of oracle queries that* $\mathcal{A}, \mathcal{B}_i$ *make. Let* q' *be (an upper bound for) the number of polynomials* $\mathcal{B}$ *supplies to* ComputeConcrete *in total.*

If $\mathcal{B}_0, \mathcal{B}_1$ *are perfectly equivalent (with respect to* $\mathcal{A}$*) and* $3q + q' + 1 \le p$*, then*

$$\begin{aligned}&\left|\Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_0^{\textsc{ComputeConcrete}}} = 1] - \Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_1^{\textsc{ComputeConcrete}}} = 1]\right| \\ \le\ & 2 \cdot (2q + q') \cdot q'/(p - q)\end{aligned}$$

Proof. From Lemma 4, we know that

$$\begin{aligned}&\left|\Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_j^{\textsc{ComputeConcrete}}} = 1] - \Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_j^{\textsc{ComputeAtomic}}} = 1]\right| \\ \le\ & (2q + q') \cdot q'/(p - q).\end{aligned}$$

Analyzing $\mathcal{B}_j^{\textsc{ComputeAtomic}}$, notice that querying ComputeAtomic with result h amounts to just querying $h \leftarrow \textsc{Tau}(i, \mathsf{f})$, i.e. there are no other side effects. Since the inputs (i, f) are distributed the same between $\mathcal{B}_0$ and $\mathcal{B}_1$ by the lemma's prerequisites, one can conclude that there is no difference between $\mathcal{B}_0$ and $\mathcal{B}_1$ in this setting. This means that

$$\left|\Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_0^{\textsc{ComputeAtomic}}} = 1] - \Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_1^{\textsc{ComputeAtomic}}} = 1]\right| = 0.$$

Overall,

$$\begin{aligned}&\left|\Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_0^{\textsc{ComputeConcrete}}} = 1] - \Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_1^{\textsc{ComputeConcrete}}} = 1]\right| \\ \le & \left|\Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_0^{\textsc{ComputeAtomic}}} = 1] - \Pr[\mathcal{A}^{\mathcal{G}\text{-}\mathtt{oSG}, \mathcal{B}_1^{\textsc{ComputeAtomic}}} = 1]\right| \\ & + 2 \cdot (2q + q') \cdot q'/(p - q) \\ = & 0 + 2 \cdot (2q + q') \cdot q'/(p - q)\end{aligned}$$

□

I Proof of Lemma 5

Proof (Lemma 5). For a representation $\mathit{Rep} = (a_j)_{j=1}^n$, we write the corresponding polynomial $\mathsf{V}(\mathit{Rep}) = \sum_{j=1}^n a_j \cdot \tau_i^{-1}(B_j) \in \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]$. Let $\mathit{Rep}[h^*] = (a_j^*)_{j=1}^n$ be the result of a $\textsc{FindRep}(i, h^*, \mathit{Ob}_{sid}, B)$ call, and let $\mathsf{V}(h^*)$ be its corresponding polynomial. Let $\mathsf{I} = \langle \mathsf{Var}_{-sid}, \tau_i^{-1}(C_i \setminus B)\rangle_{\mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm 1}]}$ be the ideal of, loosely speaking, *unobservable* polynomials, which is formed by variables of other sessions $\mathit{sid}' \neq \mathit{sid}$ and generators $g \in C_i$ not supplied as basis input via B. The assertion in Line 58 is equivalent to checking that $\mathsf{V}(\mathit{Rep}[h^*]) = \tau_i^{-1}(h^*) \mod \mathsf{I}$.

We now argue that this check never fails. For this, we consider the following claim.

Claim 3. *Whenever* $\mathit{Rep}[h]$ *is set to some value (other than the initial assignment in Line 5), it holds that* $\mathsf{V}(\mathit{Rep}[h]) = \tau_i^{-1}(h) \mod \mathsf{I}$*.*

First, note that when assigning the representations for the basis elements in Line 6, we have by definition $\mathsf{V}(\mathit{Rep}[B_j]) = \tau_i^{-1}(B_j)$. The other assignment is in Line 8 when processing the observation $\mathit{ob} = (\textsc{Op}, i, g_1, g_2, a_1, a_2, h) \in \mathit{Ob}_{sid}$. If we assume (for now) that $\mathsf{V}(\mathit{Rep}[g_j]) = \tau_i^{-1}(g_j) \mod \mathsf{I}$ $(j \in \{1, 2\})$, then the result $\mathit{Rep}[h] \leftarrow a_1 \cdot \mathit{Rep}[g_1] + a_2 \cdot \mathit{Rep}[g_2]$ also fulfills $\mathsf{V}(\mathit{Rep}[h]) = \mathsf{V}(a_1 \cdot \mathit{Rep}[g_1] + a_2 \cdot \mathit{Rep}[g_2]) = a_1 \cdot \mathsf{V}(\mathit{Rep}[g_1]) + a_2 \cdot \mathsf{V}(\mathit{Rep}[g_2]) = a_1 \cdot \tau_i^{-1}(g_1) + a_2 \cdot \tau_i^{-1}(g_2) = \tau_i^{-1}(h) \mod \mathsf{I}$.

It remains to argue that indeed, $\mathsf{V}(\mathit{Rep}[g_j]) = \tau_i^{-1}(g_j)$ $(j \in \{1, 2\})$. For this, consider the following case distinction about how g_j *first* appeared in $\mathcal{G}\text{-}\mathtt{oSG}$.

- $g_j = \tau_i(1)$ is the canonical generator. We then have two cases.
  - $g_j \in B$ has been supplied as basis element, in which case $Rep[g_j]$ has been set correctly in Line 6.
  - $g_j \notin B$ was not supplied as a basis, so its zero default value is correct: We have that $g_j \in C_i \setminus B$ and so by definition of I, we have $\tau_i^{-1}(g_j) = 0 \mod \mathsf{I}$.
- g_j was first queried to $\textsc{Touch}_{sid'}$ (either directly or indirectly via $\textsc{Op}, \textsc{Pair}$). This means that $\tau_i(g_j) \in \mathsf{Var}_{sid'}$ is simply a formal variable for session sid'. There are two cases.
  - $sid' = sid$ is the caller's session ID. Then as above, $g_j \in B$ and $Rep[g_j]$ is correctly set in Line 6, or $g_j \in C_i \setminus B$, meaning the default value of zero is correct.
  - $sid' \neq sid$ is a foreign session. Then by definition of I, we have $\tau_i^{-1}(g_j) = 0 \mod \mathsf{I}$ as required.
- g_j was first seen as the result of a $\textsc{ComputeSymbolic}_{sid'}$ query. Similarly to $\textsc{Touch}$,
  - if $sid' = sid$, then g_j is either part of the basis B or correctly zero.
  - If $sid' \neq sid$, then g_j belongs to a foreign session, meaning $\tau_i^{-1}(g_j) \in \langle \mathsf{Var}_{-sid} \rangle$, and hence $\tau_i^{-1}(g_j) = 0 \mod \mathsf{I}$.
- g_j was first seen as the result of a $\textsc{Op}_{sid'}$ query. We distinguish two cases.
  - During that $\textsc{Op}$ query, the observation ob' of result g_j was added to the observation list Ob_{sid}. In this case, ob' must have been processed earlier than ob, setting $Rep[g_j]$ correctly in Line 8 (argued inductively).
  - During that $\textsc{Op}$ query, *no* observation was added to Ob_{sid}. This implies that $sid' \neq sid$ (otherwise the observation is always added), and $\tau_i^{-1}(g_j) \in \mathsf{Legal}_{sid'}$ (otherwise an observation is added). This means that $\tau_i^{-1}(g_j) \in \langle \mathsf{Var}_{sid'} \rangle \subseteq \langle \mathsf{Var}_{-sid} \rangle$, and hence $\tau_i^{-1}(g_j) = 0 \mod \mathsf{I}$, i.e. the default zero is correct.

This concludes the argument that when Line 8 is executed, $Rep[g_1], Rep[g_2]$ are already correctly set, meaning that $Rep[h]$ is correctly set.

Finally, if $Rep[h^*]$ was set during the execution of $\textsc{FindRep}$, then it was set correctly (Claim 3). If $Rep[h^*]$ was not set during the execution of $\textsc{FindRep}$, then the same arguments above for g_j apply to h^* and show that the default zero must have been a correct value for h^*.

J Proof of supporting claims for Theorem 1

We now prove that each transition only incurs a negligible loss.

Claim 4. *Hybrids H_0 and H_1 are indistinguishable. Concretely,*

$$|\Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\mathtt{wNIZK},\mathcal{Z},\mathcal{S}_{\mathtt{G16}},\mathcal{G}\text{-}\mathtt{oGG}}(\lambda, z) = 1] - \Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\mathtt{wNIZK},\mathcal{Z},\mathcal{S}_{\mathtt{G16}},\mathcal{G}\text{-}\mathtt{oSG}}(\lambda, z) = 1]| \\ \leq \frac{9q_1^2 + 3q_1}{p}$$

where $q_1 = m + 3d + 6 + q_{\mathcal{Z}} + 3q_{\mathcal{P}} + (\ell + 6)q_{\mathcal{V}}$.

Proof: We count the number of queries made to $\mathcal{G}\text{-}\mathtt{oGG}/\mathcal{G}\text{-}\mathtt{oSG}$:
- Initial generation of CRS triggers at most $m + 3d + 4$ queries to $\textsc{Op}$ and 2 queries to $\textsc{Touch}$.
- $\mathcal{Z}$ queries $\mathcal{G}\text{-}\mathtt{oGG}/\mathcal{G}\text{-}\mathtt{oSG}$ at most $q_{\mathcal{Z}}$ times.
- Each invocation of $\textsc{Prove}$ triggers at most 3 queries to $\textsc{Op}$ via $\mathcal{S}_{\mathtt{G16}}.\textsc{Simulate}$

– Each invocation of Verify triggers at most $\ell + 2$ queries to Op and 4 queries to Pair via $\mathcal{S}_{\texttt{G16}}$.Extract.

In total, at most $q_1 = m + 3d + 6 + q_{\mathcal{Z}} + 3q_{\mathcal{P}} + (\ell + 6)q_{\mathcal{V}}$ are made during an execution of each hybrid. Plugging q_1 into Lemma 3, we obtain the claimed loss. ■

Claim 5. *Hybrids H_2 and H_1 are indistinguishable. Concretely,*

$$
\begin{aligned}
&|\Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\texttt{wNIZK},\mathcal{Z},\mathcal{S}_{\texttt{G16}},\mathcal{G}\text{-}\texttt{oSG}}(\lambda, z) = 1] - \Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\texttt{wNIZK}',\mathcal{Z},\mathcal{S}_{\texttt{G16}},\mathcal{G}\text{-}\texttt{oSG}}(\lambda, z) = 1]| \\
\leq&(2q_2 + q_2') \cdot \frac{3q_2'}{2p} + \frac{(9q_2^2 + 3q_2)d_2}{p-1} + \frac{6q_{\mathcal{Z}}}{p}
\end{aligned}
$$

where $q_2 = q_1$ (see Claim 4), $q_2' = m + 3d + 4 + 3q_{\mathcal{P}}$ and $d_2 = 2d - 1$.

Proof: First, let us consider the case where the Verify interface receives as input (x, π) such that x was previously queried to the Prove interface and Prove responded with π. In H_1, $\mathcal{F}$-wNIZK.Verify$_{sid}(x, \pi)$ always returns 1 since (x, π) is guaranteed to exist in the table T. In H_2, $\mathcal{F}$-wNIZK$'$.Verify$_{sid}(x, \pi)$ also returns 1 since (x, π) was simulated by $\mathcal{S}_{\texttt{G16}}$ in such a way that it passes the verification condition. Hence, the view of $\mathcal{Z}$ is identical in H_1 and H_2.

Second, we look at the case where the Verify interface receives as input (x, π) such that x was previously queried to the Prove interface and Prove responded with $\pi' \neq \pi$. In H_1, $\mathcal{F}$-wNIZK.Verify$_{sid}(x, \pi)$ outputs 1 if and only if the verification equation is satisfied, thanks to the additional check at Line 9 of $\mathcal{F}$-wNIZK. In H_2, $\mathcal{F}$-wNIZK$'$.Verify$_{sid}(x, \pi)$ also outputs 1 if and only if the verification equation is satisfied by definition. Hence, the view of $\mathcal{Z}$ is identical in H_1 and H_2.

We now look at the case where the Verify interface receives as input (x, π) such that x was never queried to the Prove interface. In H_1, $\mathcal{F}$-wNIZK.Verify$_{sid}(x, \pi)$ outputs 1 if only if $\mathcal{S}_{\texttt{G16}}$.Extract$_{sid}(x, \pi)$ successfully outputs w such that $(x, w) \in \mathcal{R}_{\texttt{QAP}}$. In H_2, $\mathcal{F}$-wNIZK$'$.Verify$_{sid}(x, \pi)$ outputs 1 if and only if the verification equation is satisfied by definition. Hence, the view of $\mathcal{Z}$ is identical in H_1 and H_2, *except if* $\mathcal{S}_{\texttt{G16}}$.Extract$_{sid}(x, \pi)$ *in* H_1 *fails to extract valid witness while* (x, π) *passes verification.* We now bound the probability that this exceptional event occurs in H_1. To this end, we introduce the following sub-hybrids:

– H_1': This is essentially a syntactically re-arranged version of H_1 except with one abort condition. In H_1', $\mathcal{G}$-oSG is extended with additional interfaces GetRnd and ComputeConcrete which can be accessed by a modified simulator $\mathcal{S}_{\texttt{G16}}'$ described in Simulator 2. Note that $\mathcal{S}_{\texttt{G16}}'$ now aborts if one of the session-specific generators is already reserved for another session.That is, Touch$_{sid}(i, g_{sid,i})$ fails to initialize a fresh variable associated with *sid* if there already exists some f such that $\tau_i(\mathsf{f}) = g_{sid,i}$. Since $\mathcal{Z}$ may define τ_i for at most 3 new group elements through each query to $\mathcal{G}$-oSG, the probability that $\mathcal{S}_{\texttt{G16}}'$ aborts is at most $6q_{\mathcal{Z}}/p$ by the union bound. The view of $\mathcal{Z}$ is identical in H_1 and H_1' unless $\mathcal{S}_{\texttt{G16}}'$ aborts.

– H_1'': This is identical to H_1' except that every invocation of ComputeConcrete is replaced with ComputeSymbolic as in the modified simulator described in $\mathcal{S}_{\texttt{G16}}''$ Simulator 3. This transition is justified by Lemma 4. Concretely, counting the number of supplied polynomials to ComputeConcrete/ComputeSymbolic:
  - Initial generation of CRS supplies at most $m + 3d + 4$ polynomials.
  - Each invocation of Prove sends at most 3 polynomials via Simulate
  - Each invocation of Verify does not trigger any query to ComputeX

  In total, at most $q_2' = m + 3d + 4 + 3q_{\mathcal{P}}$ polynomials are supplied during an execution of each hybrid. Moreover, the degrees of supplied polynomials are upper-bounded by $d_2 = 2d - 1$. The total number of queries q_2 to $\mathcal{G}$-oSG is the same as q_1 of the previous claim. Plugging q_2, q_2' and d_2 into Lemma 4 and accounting for the loss incurred by the abort condition of H_1', we obtain the claimed loss. (As we shall next, there won't be any more loss during the rest of the analysis.)

– H_1''': This is identical to H_1'' except that $\mathcal{G}$-$\mathtt{oSG}$ is extended with an additional $\textsc{GetRep}_{sid}$ interface and every invocation of $\textsc{FindRep}$ is replaced with $\textsc{GetRep}_{sid}$ as in the modified simulator $\mathcal{S}_{\mathtt{G16}}'''$ described in Simulator 4. This transition is justified by Lemma 5 and incurs no loss.

We now perform weak SE analysis of Groth16 in a purely symbolic manner. Since every random exponent $x, \alpha, \beta, \gamma, \delta$ sampled by $\mathcal{S}_{\mathtt{G16}}'''$ is now treated as a formal symbol, the proof $\pi = (A, B, C)$ output by $\mathcal{Z}$ can be expressed as:

$$\begin{aligned}
\tau^{-1}(A) = p_A(\mathsf{Var}_{-sid}) + \mathsf{X}_{sid,1}\Big(& A_\alpha \mathsf{X}_\alpha + A_\beta \mathsf{X}_\beta + A_\delta \mathsf{X}_\delta + A_x(\mathsf{X}_x) + A_h(\mathsf{X}_x)t(\mathsf{X}_x)\mathsf{X}_\delta^{-1} \\
&+ \sum_{i=0}^{\ell} A_i q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x)\mathsf{X}_\gamma^{-1} + \sum_{i=\ell+1}^{m} A_i q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x)\mathsf{X}_\delta^{-1} \\
&+ \sum_{j=1}^{Q} A_{\mu^{(j)}} \mathsf{X}_\mu^{(j)} + \sum_{j=1}^{Q} A_{C^{(j)}}(\mathsf{X}_{\mu^{(j)}}\mathsf{X}_{\nu^{(j)}} - \mathsf{X}_\alpha \mathsf{X}_\beta - \sum_{i=0}^{\ell} a_i^{(j)} q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x))\mathsf{X}_\delta^{-1}\Big) \\
\tau^{-1}(B) = p_B(\mathsf{Var}_{-sid}) + \mathsf{X}_{sid,2}\Big(& B_\beta \mathsf{X}_\beta + B_\gamma \mathsf{X}_\gamma + B_\delta \mathsf{X}_\delta + B_x(\mathsf{X}_x) + \sum_{j=1}^{Q} B_{\nu^{(j)}} \mathsf{X}_{\nu^{(j)}}\Big) \\
\tau^{-1}(C) = p_C(\mathsf{Var}_{-sid}) + \mathsf{X}_{sid,1}\Big(& C_\alpha \mathsf{X}_\alpha + C_\beta \mathsf{X}_\beta + C_\delta \mathsf{X}_\delta + C_x(\mathsf{X}_x) + C_h(\mathsf{X}_x)t(\mathsf{X}_x)\mathsf{X}_\delta^{-1} \\
&+ \sum_{i=0}^{\ell} C_i q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x)\mathsf{X}_\gamma^{-1} + \sum_{i=\ell+1}^{m} C_i q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x)\mathsf{X}_\delta^{-1} \\
&+ \sum_{j=1}^{Q} C_{\mu^{(j)}} \mathsf{X}_\mu^{(j)} + \sum_{j=1}^{Q} C_{C^{(j)}}(\mathsf{X}_{\mu^{(j)}}\mathsf{X}_{\nu^{(j)}} - \mathsf{X}_\alpha \mathsf{X}_\beta - \sum_{i=0}^{\ell} a_i^{(j)} q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x))\mathsf{X}_\delta^{-1}\Big)
\end{aligned}$$

where A_h, C_h are univariate polynomials of degree $d-2$, A_x, B_x, C_x are univariate polynomials of degree $d-1$, Var_{-sid} is a vector of foreign variables (as defined in Functionality 4), and p_A, p_B, p_C are multivariate polynomials, respectively. Note that $\{C_i\}_{i=\ell+1}^{m}$ is the candidate witness returned by $\textsc{GetRep}_{sid}$. Our goal is show that $(x, w) = (\{a_i\}_{i=1}^{\ell}, \{C_i\}_{i=\ell+1}^{m}) \in \mathcal{R}_{\mathtt{QAP}}$ whenever x and $\pi = (A, B, C)$ pass verification.

First, the fact that A, B, C satisfy the verification condition implies:

$$\begin{aligned}
&\tau^{-1}(A) \cdot \tau^{-1}(B) \\
&\equiv \tau^{-1}(C) \cdot (\mathsf{X}_{sid,2}\mathsf{X}_\delta) + \mathsf{X}_{sid,1}\mathsf{X}_{sid,2}\left(\mathsf{X}_\alpha \mathsf{X}_\beta + \sum_{i=0}^{\ell} a_i q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x)\right)
\end{aligned}$$

Focusing on the terms containing the monomial $\mathsf{X}_{sid,1}\mathsf{X}_{sid,2}$, we have that

$$\begin{aligned}
&(A_\alpha \mathsf{X}_\alpha + A_\beta \mathsf{X}_\beta + \ldots)(B_\beta \mathsf{X}_\beta + B_\gamma \mathsf{X}_\gamma + \ldots) \\
&\equiv \mathsf{X}_\delta (C_\alpha \mathsf{X}_\alpha + C_\beta \mathsf{X}_\beta \ldots) + \mathsf{X}_\alpha \mathsf{X}_\beta + \sum_{i=0}^{\ell} a_i q_i(\mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_x) \qquad (1)
\end{aligned}$$

This equation is identical to the one analyzed in Theorem 1 of [BKSV21], where they symbolically prove weak SE of Groth16 in the stand-alone setting. Thus, we only provide a sketch following their proof. First, [BKSV21] shows that $A_{C^{(j)}} = 0$ for all $j \in [Q]$ by comparing LHS and RHS of (1). Then they prove that only one of the following cases holds: (1) $A_{\mu^{(j)}} = B_{\nu^{(j)}} = C_{\mu^{(j)}} = C_{C^{(j)}} = 0$ for all $j \in [Q]$ implying that no simulated proof is used to construct the forged proof $\pi = (A, B, C)$, or (2) there exists some $k \in [Q]$ such that

$A_{\mu^{(k)}}$, $B_{\nu^{(k)}}$, $C_{\mu^{(k)}}, C_{C^{(k)}}$ and $A_{\mu^{(j)}} = B_{\nu^{(j)}} = C_{\mu^{(j)}} = C_{C^{(j)}} = 0$ for $j \neq k$[17], implying that only the kth simulated proof and CRS are used to construct the forged proof. In Case (1), one can invoke the plain knowledge soundness analysis of [Gro16, Theorem 1], guaranteeing that $\{C_i\}_{i=\ell+1}^{m}$ is valid QAP witness corresponding to the received statement $x = \{a_i\}_{i=1}^{\ell}$. In Case (2), [BKSV21] shows that the received statement $x = \{a_i\}_{i=1}^{\ell}$ is identical to the kth statement $x^{(k)} = \{a_i^{(k)}\}_{i=1}^{\ell}$, meaning that the forged proof π is merely a mauled version of kth simulated proof $\pi^{(k)}$. In this case, the simulator does not need to extract valid witness as we described before.

■

Claim 6. *Hybrids H_2 and H_3 are indistinguishable. Concretely,*

$$|\Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\mathtt{wNIZK}',\mathcal{Z},\mathcal{S}_{\mathtt{G16}},\mathcal{G}\text{-}\mathtt{oSG}}(\lambda, z) = 1] - \Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\mathtt{wNIZK}'',\mathcal{Z},\mathcal{S}_{\mathtt{G16}},\mathcal{G}\text{-}\mathtt{oSG}}(\lambda, z) = 1]| \\ \leq \frac{6 \cdot (2q_3 + q_3') \cdot q_3'}{p} + \frac{12 q_{\mathcal{Z}}}{p}$$

where $q_3 = q_1$ (see Claim 4) and $q_3' = m + 3d + 4 + (m - \ell + 4d + 1)q_{\mathcal{P}}$.

Proof: As $\mathcal{S}_{\mathtt{G16}}$ in H_2 follows the perfect ZK simulation routine of [Gro16], the distribution of each simulated (A, B, C) in H_2 is identical to that in H_3. Note that a sequence of group operations associated with leading to each simulated/honestly generated proof is different i.e. $A = [\mu]_{sid,1}$ in H_2 whereas $A = [\alpha]_{sid,1} + r[\delta]_{sid,1} + \ldots$ in H_3. To argue this change is unnoticed by the environment, we would like to invoke Lemma 7. To this end, we introduce the following sub-hybrids:

- H_2': Similar to H_1' in Claim 5, this is a syntactically re-arranged version of H_2 except with one abort condition: the modified version of $\mathcal{S}_{\mathtt{G16}}$ aborts if touching the session-specific group generators fails (by checking whether their representation is already defined or not). Moreover, Simulate invokes ComputeConcrete to obtain simulated A, B, C.
- H_3': Similar to H_1' in Claim 5, this is a syntactically re-arranged version of H_3 except with one abort condition: the modified version of $\mathcal{S}_{\mathtt{G16}}$ aborts if touching the session-specific group generators fails. Moreover, Prove invokes ComputeConcrete to obtain honestly computed A, B, C after initializing X_r and X_s via GetRnd.

The loss incurred when transitioning to H_2' from H_2 is at most $6q_{\mathcal{Z}}/p$. The same loss applies when transitioning to H_3' from H_3. Regarding Simulate of H_2' as $\mathcal{B}_0$ and Prove of H_3' as $\mathcal{B}_1$, respectively, they indeed satisfy the perfectly equivalence condition as required by Lemma 7 because the joint distribution of $(\tau_1^{-1}(A), \tau_2^{-1}(B), \tau_1^{-1}(C))$ output by both algorithms is identical. To derive the concrete loss, let us count the number of supplied polynomials to ComputeConcrete:

- Initial generation of CRS supplies at most $m + 3d + 4$ polynomials.
- Each invocation of Prove sends at most $m - \ell + 4d + 1$ polynomials (bounded by the number of polynomials sent in H_3').
- Each invocation of Verify does not trigger any query

In total, at most $q_3 = m + 3d + 4 + (m - \ell + 4d + 1)q_{\mathcal{P}}$ polynomials are supplied during an execution of each hybrid. The total number of queries q_3 to $\mathcal{G}$-$\mathtt{oSG}$ is the same as q_1 of the previous claim. Plugging q_3 and q_3' into Lemma 7 and accounting for the loss incurred by the abort condition of H_2' and H_3', we obtain the claimed loss. ■

[17] Although [BKSV21] does not explicitly mention $A_{\mu^{(j)}}$ and $C_{\mu^{(j)}}$ (denoted by $A_{8,j}$ and $C_{8,j}$ in their proof, respectively) are 0, we can indeed confirm they are 0 by looking at the relevant constraints of (1). According to their analysis $B_{\nu^{(k)}} \neq 0$. Then looking at the term involving $\mathsf{X}_{\mu^{(i)}}\mathsf{X}_{\nu^{(j)}}$ for $i \neq j$, we have that $A_{\mu^{(i)}} B_{\nu^{(j)}} = 0$, implying $A_{\mu^{(i)}} = 0$ for $i \neq k$. Looking at the term involving $\mathsf{X}_{\mu^{(i)}}\mathsf{X}_{\delta}$, we also have that $A_{\mu^{(i)}} B_{\delta} - C_{\mu^{(i)}} = 0$ for all $i \in [Q]$. Thus, $C_{\mu^{(i)}} = 0$ for $i \neq k$.

Claim 7. *Hybrids H_3 and H_4 are indistinguishable. Concretely,*

$$\begin{aligned}&|\Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\mathtt{wNIZK}'',\mathcal{Z},\mathcal{S}_{\mathtt{G16}},\mathcal{G}\text{-}\mathtt{oSG}}(\lambda,z)=1]-\Pr[\mathsf{EXEC}_{\mathcal{F}\text{-}\mathtt{wNIZK}'',\mathcal{Z},\mathcal{S}_{\mathtt{G16}},\mathcal{G}\text{-}\mathtt{oGG}}(\lambda,z)=1]|\\&\leq\frac{9q_4^2+3q_4}{p}\end{aligned}$$

where $q_4 = m + 3d + 6 + 3q_{\mathcal{Z}} + (m - \ell + 4d + 1)q_{\mathcal{P}} + (\ell + 6)q_{\mathcal{V}}$.

Proof: We count the number of queries made to $\mathcal{G}$-$\mathtt{oGG}$/$\mathcal{G}$-$\mathtt{oSG}$:

- Initial generation of CRS triggers at most $m + 3d + 4$ queries to OP and 2 queries to TOUCH.
- $\mathcal{Z}$ queries $\mathcal{G}$-$\mathtt{oGG}$/$\mathcal{G}$-$\mathtt{oSG}$ at most $q_{\mathcal{Z}}$ times.
- Each invocation of PROVE triggers at most $m - \ell + 4d + 1$ queries to OP.
- Each invocation of VERIFY triggers at most $\ell + 2$ queries to OP and 4 queries to PAIR via $\mathcal{F}$-$\mathtt{wNIZK}''$.EXTRACT.

In total, at most $q_4 = m + 3d + 6 + 3q_{\mathcal{Z}} + (m - \ell + 4d + 1)q_{\mathcal{P}} + (\ell + 6)q_{\mathcal{V}}$ are made during an execution of each hybrid. Plugging q_4 into Lemma 3, we obtain the claimed loss. ■

Functionality 20: $\mathcal{F}$-$\texttt{wNIZK}'$ (used in hybrid H_2)

$\mathcal{F}$-$\texttt{wNIZK}'$ is parameterized by $\mathcal{R}_{\texttt{QAP}}$, and runs with parties $\mathcal{P}_1, \ldots, \mathcal{P}_N$ and an ideal process adversary $\mathcal{S}_{\texttt{G16}}$. Moreover, it has direct access to $\mathcal{G}$-$\texttt{oSG}$. The group operations happening inside Verify are carried out via the corresponding wrapper interfaces of $\mathcal{S}_{\texttt{G16}}$. It stores proof table T which is initially empty.

```
Init_sid()
  1: $T \leftarrow []$ // Empty table

Prove_sid(x, w)
  1: if $(x, w) \notin \mathcal{R}$ then return $\perp$
  2: $\pi \leftarrow \mathcal{S}_{\texttt{G16}}.\textsc{Simulate}_{sid}(x)$
  3: $T \leftarrow T \cup (x, \pi)$
  4: return $\pi$

Verify_sid($x = \{a_i\}_{i=1}^{\ell}, \pi = (A, B, C)$)
  1: $\sigma \leftarrow \mathcal{S}_{\texttt{G16}}.\textsc{GetCRS}_{sid}()$
  2: $C_{\text{pub}} \leftarrow \left[\sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x)\gamma^{-1}\right]_{sid,1}$
  3: return $A \cdot B = C_{\text{pub}} \cdot [\gamma]_{sid,2} + C \cdot [\delta]_{sid,2} + [\alpha]_{sid,1} \cdot [\beta]_{sid,2}$
```

Functionality 21: $\mathcal{F}$-$\texttt{wNIZK}''$ (used in hybrid H_3)

$\mathcal{F}$-$\texttt{wNIZK}''$ is parameterized by $\mathcal{R}_{\texttt{QAP}}$, and runs with parties $\mathcal{P}_1, \ldots, \mathcal{P}_N$ and an ideal process adversary $\mathcal{S}_{\texttt{G16}}$. Moreover, it has direct access to $\mathcal{G}$-$\texttt{oSG}$. The group operations happening inside Prove and Verify are carried out via the corresponding interfaces of $\mathcal{G}$-$\texttt{oSG}$. It stores proof table T which is initially empty.

```
Init_sid()
  1: $T \leftarrow []$ // Empty table

Prove_sid($x = \{a_i\}_{i=1}^{\ell}, w = \{a_i\}_{i=\ell+1}^{m}$)
  1: if $(x, w) \notin \mathcal{R}_{\texttt{QAP}}$ then return $\perp$
  2: $\sigma \leftarrow \mathcal{S}_{\texttt{G16}}.\textsc{GetCRS}_{sid}()$
  3: $r, s \xleftarrow{\$} \mathbb{Z}_p$
  4: Compute $h \in \mathbb{F}^{d-2}[X]$ such that $ht = (\sum_{i=0}^{m} a_i u_i)(\sum_{i=0}^{m} a_i v_i) - (\sum_{i=0}^{m} a_i w_i)$
  5: $A := [a]_{sid,1} \leftarrow \left[\sum_{i=0}^{m} a_i u_i(x) + \alpha + r\delta\right]_{sid,1}$
  6: $B := [b]_{sid,2} \leftarrow \left[\sum_{i=0}^{m} a_i v_i(x) + \beta + s\delta\right]_{sid,2}$
  7: $C := [c]_{sid,1} \leftarrow \left[\sum_{i=\ell+1}^{m} a_i q_i(\alpha, \beta, x)\delta^{-1} + h(x)t(x)\delta^{-1} + sa + rb - rs\delta\right]_{sid,1}$
  8: return $(A, B, C)$

Verify_sid($x = \{a_i\}_{i=1}^{\ell}, \pi = (A, B, C)$)
  1: $\sigma \leftarrow \mathcal{S}_{\texttt{G16}}.\textsc{GetCRS}_{sid}()$
  2: $C_{\text{pub}} \leftarrow \left[\sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x)\gamma^{-1}\right]_{sid,1}$
  3: return $A \cdot B = C_{\text{pub}} \cdot [\gamma]_{sid,2} + C \cdot [\delta]_{sid,2} + [\alpha]_{sid,1} \cdot [\beta]_{sid,2}$
```

Simulator 1: $\mathcal{S}_{\mathtt{G16}}$

The simulator stores state:

- Ob_{sid}, Ob' initially empty lists
- σ Labels for simulated common reference string
- td Trapdoor for σ

GETCRS, OP, PAIR, TOUCH, CANONICALGEN, OBSERVE are to be called by the environment, while SIMULATE and EXTRACT are to be called by $\mathcal{F}$-wNIZK. Note that the interfaces OP, PAIR, TOUCH, CANONICALGEN, OBSERVE are wrappers of the corresponding methods of $\mathcal{G}$-oGG. We define these so that $\mathcal{S}_{\mathtt{G16}}$ can keep track of all the group operations happening inside the current session.

```
INIT_sid()  // Invoked only upon creation
    Run the code for $\mathcal{F}$-CRS.INIT_sid(). Store $\sigma$ as CRS and $td = (x, \alpha, \beta, \delta)$ as a simulation trapdoor, respectively.

GETCRS_sid()
    return $\sigma$

OP_sid($i, g_1, g_2, a_1, a_2$)
    $h \leftarrow \mathcal{G}$-oGG.OP_sid($i, g_1, g_2, a_1, a_2$)
    assert $h \neq \bot$
    UPDATEOB_sid((OP, $i, g_1, g_2, a_1, a_2, h$))
    return $h$

PAIR_sid($g_1, g_2$)
    $h \leftarrow \mathcal{G}$-oGG.PAIR_sid($g_1, g_2$)
    assert $h \neq \bot$
    UPDATEOB_sid((PAIR, t, $g_1, g_2, h$))
    return $h$

TOUCH_sid($i, g$)
    return $\mathcal{G}$-oGG.TOUCH_sid($i, g$)

CANONICALGEN_sid($i$)
    return $\mathcal{G}$-oGG.CANONICALGEN_sid($i$)

OBSERVE_sid()
    return $\mathcal{G}$-oGG.OBSERVE_sid()

UPDATEOB_sid(tuple)
    $Ob^* \leftarrow \mathcal{G}$-oGG.OBSERVE_sid()
    $Ob_{sid} \leftarrow Ob_{sid} : (Ob^* \setminus Ob')$ : tuple
    $Ob' \leftarrow Ob^*$  // Stash the current state of observation list stored in $\mathcal{G}$-oGG

SIMULATE_sid($x = \{a_i\}_{i=1}^{\ell}$)
    $\mu, \nu \xleftarrow{\$} \mathbb{Z}_p$
    $A \leftarrow [\mu]_{sid,1}$
    $B \leftarrow [\nu]_{sid,2}$
    $C \leftarrow [(\mu\nu - \alpha\beta - \sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x))\delta^{-1}]_{sid,1}$  // This operation requires the knowlege of $td$.
    return $(A, B, C)$

EXTRACT_sid($x = \{a_i\}_{i=1}^{\ell}, \pi = (A, B, C)$)
    $C_{\text{pub}} \leftarrow [\sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x)\gamma^{-1}]_{sid,1}$
    if $A \cdot B \neq C_{\text{pub}} \cdot [\gamma]_{sid,2} + C \cdot [\delta]_{sid,2} + [\alpha]_{sid,1} \cdot [\beta]_{sid,2}$ then
        return junk  // No need to extract if $(x, \pi)$ is invalid
    UPDATEOB_sid()  // Complete the fully ordered observation list
    $g \leftarrow \mathcal{G}$-oGG.CANONICALGEN_sid(1)
    $B \leftarrow (g, [1]_{sid,1}, [\{q_i(\alpha, \beta, x)\delta^{-1}\}_{i=\ell+1}^{m}]_{sid,1})$
    $(\cdot, \cdot, \{C_{q_i}\}_{i=\ell+1}^{m}) \leftarrow$ FINDREP($1, C, Ob_{sid}, B$)  // See Function 1
    $w \leftarrow \{C_{q_i}\}_{i=\ell+1}^{m}$
    if $(x, w) \in \mathcal{R}_{\mathtt{QAP}}$ then
        return $w$
    else
        return maul  // If extraction fails but the proof verifies, mark it mauled. The functionality will eventually return 1 if $x$ was previously queried.
```

Simulator 2: $\mathcal{S}'_{\texttt{G16}}$

The simulator stores state:

- Ob_{sid} initially empty list
- σ Labels for simulated common reference string
- td Trapdoor for σ

GetCRS, Op, Pair, Touch, CanonicalGen, Observe, Extract are identical to those of $\mathcal{S}_{\texttt{G16}}$ except they call $\mathcal{G}$-$\texttt{oSG}$. We overload the bracket notations within $\mathcal{S}'_{\texttt{G16}}$ as follows, assuming that symbolic variables $\mathsf{X}_x, \mathsf{X}_y, \ldots$ are obtained through GetRnd:

$$[f(x, y, \ldots)]_{sid,i} := \mathcal{G}\text{-}\texttt{oSG}.\textsc{ComputeConcrete}(i, g_{sid,i}, f(\mathsf{X}_x, \mathsf{X}_y, \ldots))$$

```
Init_sid()   // Invoked only upon creation
1: for $i = 1, 2$ do
2:     $g_{sid,i} \xleftarrow{\$} S_i$
3:     $\mathcal{G}$-oSG.Touch$_{sid}(i, g_{sid,i})$
4:     if $g_{sid,i} \in \mathrm{im}(\tau_i)$ then abort
5: $td := (\mathsf{X}_x, \mathsf{X}_\alpha, \mathsf{X}_\beta, \mathsf{X}_\gamma, \mathsf{X}_\delta) \leftarrow \mathcal{G}$-oSG.GetRnd$_{sid}()$ // Abusing notation
6: $\sigma_1 \leftarrow [\alpha, \beta, \delta, \{x^i\}_{i=0}^{d-1}, \{q_i(\alpha, \beta, x)\gamma^{-1}\}_{i=0}^{\ell}, \{q_i(\alpha, \beta, x)\delta^{-1}\}_{i=\ell+1}^{m}, \{x^i t(x)\delta^{-1}\}_{i=0}^{d-2}]_{sid,1}$
7: $\sigma_2 \leftarrow [\beta, \gamma, \delta, \{x^i\}_{i=0}^{d-1}]_{sid,2}$
8: $\sigma \leftarrow (\sigma_1, \sigma_2)$

Simulate_sid($x = \{a_i\}_{i=1}^{\ell}$)
34: $\mathsf{X}_\mu, \mathsf{X}_\nu \leftarrow \mathcal{G}$-oSG.GetRnd$_{sid}()$
35: $A \leftarrow [\mu]_{sid,1}$
36: $B \leftarrow [\nu]_{sid,2}$
37: $C \leftarrow [(\mu\nu - \alpha\beta - \sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x))\delta^{-1}]_{sid,1}$
38: return $(A, B, C)$
```

Simulator 3: $\mathcal{S}''_{\texttt{G16}}$

Identical to $\mathcal{S}'_{\texttt{G16}}$ except that the bracket notations are overloaded as follows:

$$[f(x, y, \ldots)]_{sid,i} := \mathcal{G}\text{-}\texttt{oSG}.\textsc{ComputeSymbolic}(i, g_{sid,i}, f(\mathsf{X}_x, \mathsf{X}_y, \ldots))$$

Simulator 4: $\mathcal{S}'''_{\texttt{G16}}$

Identical to $\mathcal{S}''_{\texttt{G16}}$ except the Extract interface

```
Extract_sid($x = \{a_i\}_{i=1}^{\ell}, \pi = (A, B, C)$)
1: $C_{\mathrm{pub}} \leftarrow [\sum_{i=0}^{\ell} a_i q_i(\alpha, \beta, x)\gamma^{-1}]_{sid,1}$
2: if $A \cdot B \neq C_{\mathrm{pub}} \cdot [\gamma]_{sid,2} + C \cdot [\delta]_{sid,2} + [\alpha]_{sid,1} \cdot [\beta]_{sid,2}$ then
3:     return junk
4: $g \leftarrow \mathcal{G}$-oSG.CanonicalGen$_{sid}(1)$
5: $B \leftarrow (g, [1]_{sid,1}, [\{q_i(\alpha, \beta, x)\delta^{-1}\}_{i=\ell+1}^{m}]_{sid,1})$
6: $(\cdot, \cdot, \{C_{q_i}\}_{i=\ell+1}^{m}) \leftarrow \mathcal{G}$-oSG.GetRep$_{sid}(1, C, B)$
7: $w \leftarrow \{C_{q_i}\}_{i=\ell+1}^{m}$
8: if $(x, w) \in \mathcal{R}_{\texttt{QAP}}$ then
9:     return $w$
10: else
11:     return maul
```