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

Jan Bobolz\textsuperscript{1}, Pooya Farshim\textsuperscript{2,3}, Markulf Kohlweiss\textsuperscript{1,4}, Akira Takahashi\textsuperscript{5}

\textsuperscript{1} University of Edinburgh, UK
\textsuperscript{2} IOG, Switzerland
\textsuperscript{3} Durham University, UK
\textsuperscript{4} IOG, UK

May 26, 2024

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.
# Table of Contents

1 Introduction ........................................................................................................... 3  
   1.1 Our contributions. ............................................................................................. 5  
   1.2 Overview of our techniques. ............................................................................. 6  
   1.3 Related work ................................................................................................... 9  
2 Preliminaries .......................................................................................................... 10  
   2.1 Notation ........................................................................................................... 10  
   2.2 Generic bilinear groups .................................................................................. 11  
   2.3 The UC framework and its execution model ................................................... 11  
   2.4 Weak NIZK functionality ............................................................................... 13  
3 The global observable generic group functionality .............................................. 13  
   3.1 Warm-up: The (strict) global generic group functionality ............................... 14  
   3.2 The (restricted) observable global generic group functionality ....................... 14  
4 Switching to symbolic groups .................................................................................. 18  
   4.1 The restricted observable global symbolic group model with perfect session ... 19  
       separation ........................................................................................................... 19  
   4.2 Extending $G$-oSG with support for symbolic analysis .................................. 21  
   4.3 Extracting discrete logarithm representations ............................................... 23  
5 UC security of Groth16 ............................................................................................ 24  
6 Composition when unobservability is required .................................................... 27  
7 Conclusion and future work ................................................................................... 30  
A Related work on UC security and SNARKs ....................................................... 35  
B FAQ ....................................................................................................................... 35  
C An overview of observability in global functionalities ....................................... 37  
D Failed attempts at the $G$-oSG functionality ..................................................... 41  
E FindRep compatible with $G_t$ ............................................................................. 45  
F Proof of Lemma 3 ................................................................................................. 46  
G Proof of the first part of Lemma 4 ....................................................................... 50  
   G.1 Indistinguishability of $\text{ComputeConcrete}$ and $\text{ComputeAtomic}$ ......... 50  
H A technical lemma for switching computation styles in $G$-oSG ............................ 52  
I Proof of Lemma 5 ................................................................................................. 53  
J Proof of supporting claims for Theorem 1 ......................................................... 54
1 Introduction

The composable treatment of modern cryptosystems measures the security of a real system relative to that of an ideal system, referred to as an ideal functionality in the popular universal composability (UC) model [Can01]. Various composition theorems then show that if these systems are ‘close’, the ideal system can be safely replaced by the real system in a wide variety of contexts.

Unfortunately, to date such composable treatments of security due to their complexity often result in complicated and less efficient protocols. This state of affairs is somewhat dissatisfying as it is exactly the simple and efficient cryptosystems proven in less composable models that are widely deployed and used in complex and unpredictable environments. This necessitates the need for analyses of their composability. A notable example is that of succinct non-interactive arguments of knowledge (SNARKs), which are seeing ever wider adoption in practice (particularly in complex blockchain protocols). Yet, SNARKs exactly fall into the gap between composition and practice: either they are analyzed under property-based definitions, or else need to be modified or compiled, which increases overheads both in proof sizes and prover/verifier complexity and prevents adoption.

SNARKs are often proven secure using some idealized resource, such as the random oracle model or the generic group model. The security proofs for SNARKs are generally done in a standalone (non-composable) manner. In particular, the knowledge extractor gets exclusive control over the idealized resource, such as the random oracle or the generic group. As a simple example, this exclusive control enables a SNARK simulator/extractor to program the random oracle $H(0)$ to some desired value (and, say, embed a trapdoor into that value). While this enables simple and elegant security proofs, these proofs do not necessarily give any guarantees about compositions of the SNARK with other systems. The conflict in that scenario is that both systems’ security proofs require exclusive access to the same idealized resource. For example, the extractors of two SNARK systems may both want to program $H(0)$ to different values. As another example, the extractor for one proof of knowledge may want to observe all random oracle queries $H(\cdot)$ in order to help extract, while the simulator for another zero knowledge proof system may want to keep its oracle queries/programming secret in order to not break the 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 a need for idealized models that are compatible with composability. For the random oracle model, this has been largely solved in the form of the UC functionality $\mathcal{G}$-roRO, the restricted observable global random oracle functionality [CJS14,CDG+18], and by its programmable version $\mathcal{G}$-rpoRO [CDG+18]. The functionality $\mathcal{G}$-roRO works like a globally accessible random oracle and is not exclusively controlled by any UC simulator. Instead, it implements a mechanism through which each UC simulator gets partial control over it, in the form of observability: The UC simulator for the protocol running in session $sid$ is able to observe all random oracle queries $H(sid, \cdot)$ prefixed with $sid$ (that are made in protocol sessions $sid' \neq sid$). Details are discussed in Appendix C. With this mechanism, the single random oracle resource can be shared among multiple UC simulators in a way that still gives simulators some power over the resource (observations, or, in the case of $\mathcal{G}$-rpoRO, programming), but in a way that composes 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 $\mathcal{G}$-roRO, they can do so with sufficient domain separation not to interfere with each other. As a result, we can prove many proof systems and SNARK constructions UC-secure in the presence of $\mathcal{G}$-roRO [LR22b,LR22a,GKO+23,CF24].

---

5 Alternatively, they are proven using a knowledge assumption or in the algebraic group model (AGM). For simplicity of exposition, we treat these as idealized resources, too. For example, we can imagine the AGM as an ideal resource that the adversary deposits discrete logarithm representations of group elements into.
In contrast to proof systems in the random oracle model, the situation for proof systems 
whose proof of knowledge extraction strategy relies on idealized group-related resources 
(meaning GGM, AGM, knowledge assumptions) is much less clear. An example for such a 
proof system is the popular Groth16 SNARK [Gro16]. While these proof systems are often 
“close” to UC security, in the sense they have the prerequisite simulation-extractability and 
straightline extraction properties, their extraction strategy is not easily compatible with 
composable frameworks such as UC. This has led to the popular strategy of applying some 
transformation to the SNARK in order to change the extraction strategy towards a more 
UC-compatible strategy. The cost of this, however, is overhead: The transformed SNARK is 
much less efficient. One has to accept a significant loss in computational efficiency [GKO+23], 
or even lose succinctness [KZM+15] (see Appendix A for a review of the state-of-the-art).

Our goal is to avoid such overhead and to prove SNARKs such as Groth16 secure in 
a composable framework as-is, using their native extraction strategy. 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 \( F\)-GG hybrid model, where \( F\)-GG (e.g., [CNPR22]) is simply 
an ideal functionality implementing a generic group.

2. Prove Groth16 secure in the UC-AGM [ABK+21], which is a composable version of the 
algebraic group model. It is implemented as a modified UC model where adversaries are 
forced to output a group element’s discrete logarithm representation in terms of input 
elements whenever they output a group element.

3. Prove Groth16 secure through [KKK21], which is a composable version of knowledge 
assumptions. It is implemented as a variant of the constructive cryptography framework 
where all nodes are forced to register a group element’s discrete logarithm representation 
in terms of input elements with a global registry whenever they output a group element.

The first option is certainly feasible and a Groth16 proof in the \( F\)-GG hybrid model 
would be considered a folklore adaptation of the standalone Groth16 generic group security 
proof [Gro16]. However, the interpretation of \( F\)-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 \textit{environment} in 
the UC-AGM is \textit{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 for verification by the ideal functionality \( F\)-NIZK 
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 our purposes. 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). As discussed in [ABK+21, Section 1.1], 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 compromises its ability to reflect arbitrary environment/attacker behavior, which is a major downside. We discuss an example in Appendix B.

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 is an open challenge that requires future investigation.

This leaves open the question of a framework for a composable group-based idealized resource, which (1) enables the modeling of multiple protocols using the same group, which (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 which (3) is suitable to prove modern SNARKs based on idealized algebraic models, such as Groth16, secure.

1.1 Our contributions.

Driven by the idea that the algebraic group model suffers from inherent composability issues, we instead turn our attention to the generic group model. We propose a new solution, which comes in the form of a new restricted observable global generic (bilinear) group functionality \( G_{oGG} \) (Section 3). Similar to its random oracle counterpart, \( G_{oGG} \) works like a globally accessible generic group, but additionally models an observability mechanism based on domain separation. \( G_{oGG} \) 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, \( G_{oGG} \) naturally features oblivious sampling, i.e. generating a group element in a way that its discrete logarithm is unknown (as in hashing into a group, e.g., [BLS01]). As observed in the literature [LPS23, BFHK23], this is an important feature of real-world (elliptic curve) groups to be reflected in an idealized model.

For protocol designers relying on \( G_{oGG} \), we provide a useful security proof framework for \( G_{oGG} \)-based proofs (Section 4), which simplifies the process through a series of lemmas that enable the kind of symbolic analysis that is core to essentially all generic group proofs.

Using our security proof framework, we prove (Section 5) that Groth16 UC-realizes the ideal (weak\(^6\)) NIZK functionality \( F_{-wNIZK} \) in the presence of \( G_{oGG} \). 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

\( ^6 \) 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 for a typical UC application. As an analogy, typical use cases of signatures only require the existential unforgeability instead of a strong one.
advantage of any environment, depending on its query complexity, the size of the group, and the size of the circuit.

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\(^7\): hashes of $(\text{sid}, x)$ belong to session $\text{sid}$, and they become observable if computed in some session $\text{sid}' \neq \text{sid}$.

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 $\text{sid}$ a random generator $g_{\text{sid}}$. Intuitively, all operations done on $g_{\text{sid}}$ or group elements derived from it belong to session $\text{sid}$. 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 $g_{\text{sid}}$ and $g_{\text{sid}'}$ in two different sessions and meaningfully operate on them. This raises the question whether cross-session operations such as $g_{\text{sid}} + g_{\text{sid}'}$ are observable, which session they belong 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 $g_{\text{sid}}$ corresponds to a formal (polynomial) variable $X_{\text{sid}}$. A group element such as $g_{\text{sid}} + g_{\text{sid}'}$ is associated with the polynomial $X_{\text{sid}} + X_{\text{sid}'}$. A group operation in protocol session $\text{sid}$ is illegal (and hence observable) if the polynomial associated to the operation’s result contains any foreign-session variables $X_{\text{sid}'}$ (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 $g_{\text{sid}} + g_{\text{sid}'} - g_{\text{sid}'}$ is incorrectly associated with both sessions $\text{sid}, \text{sid}'$, which causes issues with too much or too little observability.

In the explanation above, every session $\text{sid}$ only has a single generator $g_{\text{sid}}$. In our proper $\mathcal{G}$-oGG (Section 3), a protocol can simply call the TOUCH operation on a random group element to declare it an additional generator for its session. Hence every session can have multiple generators $g_{\text{sid,1}}, g_{\text{sid,2}}, \ldots$ and the observability mechanism generalizes naturally (the explanation above applies verbatim to the multiple-generator setting).

A note on cross-session group element use. Note that in the $\mathcal{G}$-oGG setup, 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 knowledge of discrete logarithms is managed in UC-AGM vs $\mathcal{G}$-oGG. In

\(^7\) For the reader unfamiliar with domain separation approaches for global UC functionalities, Appendix C offers an explanation.
the UC-AGM, 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 $G$-oGG, there is no burden on the environment/adversary to keep track of representations. The knowledge of discrete logarithm representations is effectively maintained by $G$-oGG through observations: certain group operations are observable, and from those observations, anyone can compute (partial\(^8\)) 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 $G$-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 allow 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\textsuperscript{+}18].) Convenitely, 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 $G$-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 $G$-oGG, we refer to the UC-GS (UC with global subroutines) framework of [BCH\textsuperscript{+}20], whose 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 $G$-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 $G$-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 modification, in the $F$-CRS-hybrid model, UC-realizes the weak NIZK functionality $F$-wNIZK in the presence of $G$-oGG (Theorem 1). To the best of our knowledge, this is the first result to establish the UC security of Groth16 with zero overhead.

\(^8\) “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 details.
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 $\pi$ into a new one $\pi^*$ 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 $\pi^*$ 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 $G_{-oSG}$ 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 a bit more detail, we introduce a “fully symbolic” counterpart of the aforementioned $G_{-oGG}$, 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 $G_{-oSG}$ accepting a negligible loss in security.

Moreover, we provide a lemma that introduces a routine which makes a 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 [BKS21] almost as it 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. $G_{-oGG}$ 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 $G_{-oGG}$ 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 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 $F$-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 $F$-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 $G$-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 $G$-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 $G$-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 $G$-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 $G_1$ via index calculus methods. As a result, the guarantees provided by the GGM are somewhat less meaningful. The semigeneric group model [JR10] addresses this weakness by modeling $G_1$ as non-generic (while $G_1, G_2$ 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 on 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 on 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 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] is blind to 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 the proof sizes 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 so-called simulation-extractability (SE) property [Sah99, DDO+01, Gro06, FKMV12]. While Groth16 and variants already have SE in the GGM/AGM [BKS21, BG18, GM17], its implications to composable security have been unclear before 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] UC-secure in the presence of G-ropR0, 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 $\text{im}(\tau) = \{ y \mid \exists x : \tau(x) = y \} \subseteq B$ and the domain $\text{dom}(\tau) = \{ x \mid \tau(x) \neq \bot \} \subseteq A$. We write “assert $\phi$” 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 $A \cdot B := \{ a \cdot b \mid a \in A, b \in B \}$. We still let $A^n = A \times A \times \cdots \times A$ to denote 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., $A \in \mathbb{Z}_p[X]$). We also consider polynomials and variable with negative degree, e.g. $2X + 3X^{-1} \in \mathbb{Z}_p[X,X^{-1}]$. Sets or maps involving scalars are generally written as $S$, if they involve polynomials, they are written as $\mathbf{S}$. For a $\mathbf{Var}$ a set of variables, we let $\mathbf{Var}^{\pm 1} := \mathbf{Var} \cup \mathbf{Var}^{-1}$, where $\mathbf{Var}^{-1}$ is a set containing the inversion of variables in $\mathbf{Var}$.

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

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

**Lemma 1 (Schwartz–Zippel).** Let $F$ be a finite field, let $\mathbf{Var} = (X_1, \ldots, X_n)$ be a list of formal variables. Let $f \in \mathbb{F}[\mathbf{Var}]$, $f \neq 0$. Then

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

where the probability is over $x_1, \ldots, x_n \in \mathbb{F}$. 10
Lemma 2 (Schwartz–Zippel for Laurent polynomials). Let $\mathbb{F}$ be a finite field of order $p > 1$, let $\text{Var} = (Y_1, \ldots, Y_n)$ be a list of formal variables. Let $f \in \mathbb{F}[\text{Var}^{\pm 1}]$ be a Laurent polynomial, $f \neq 0$. Then

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

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 $y_1, \ldots, y_n \leftarrow \mathbb{F}^*$.

2.2 Generic bilinear groups

Philosophically, the generic group model represents an idealization of a 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 major forms: One due to Shoup and Nachaev [Nec94, Sho97] that idealizes element encodings as random strings, and the other due to Maurer [Mau05] that treat group elements as abstract handles. (See also [Zha22] for a more modern perspective and comparisons.) In this work we focus on Shoup’s model adopted to the case of bilinear groups.

The bilinear generic-group model is parameterized by $(p, S_1, S_2, S_3)$, 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 $S_3$. All parties, honest or otherwise, are given oracle access to three random injections $\tau_i \leftarrow \text{Inj}(\mathbb{Z}_p, S_i)$ for $i = 1, 2, t$ as well as $(\tau_1(1), \tau_2(1), \tau_3(1))$.

In this model, parties also get oracle access to three compatibly defined group operation oracles which invert a given element via $\tau_i^{-1}$, perform addition over $\mathbb{Z}_p$, and re-encode via $\tau_i$. Finally, a pairing operation allows “multiplying” two elements, one in $S_1$ and the other in $S_2$, via inversions under $\tau_1$ and $\tau_2$ respectively, multiplication over $\mathbb{Z}_p$, and encoding via $\tau_3$.

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 $\tau_1$ and $\tau_2$ 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 $\tau_2^{-1}$ and re-encoding under $\tau_1$. 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 $(\tau_1(1), \tau_2(1), \tau_3(1))$ 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 [Can20]. Here, we provide a summary of [Can20] 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 ITM itself is just a static piece of code, for each *session identifier* \( \text{id} \in \mathbb{N} \), we consider a collection of *ITM instances (ITI)* sharing the same \( \text{id} \). 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 \( \Pi \) 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 \( \Pi \) and receiving their outputs. An *identity bound* \( \xi \) places restrictions on whom \( \mathcal{Z} \) can provide input to (e.g., to ensure the environment cannot make calls to subroutines of \( \Pi \) on behalf of \( \Pi \)). The execution terminates when the environment finally terminates with an output 0 or 1.

During an execution of \( \Pi \), 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 \( \text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z) \) the distribution of a binary output by \( \mathcal{Z} \) after an execution of \( \Pi \) 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 \( \{\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*} \).

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 \( \lambda_0 \in \mathbb{N} \) such that for all \( \lambda > \lambda_0 \) and all \( z \in \cup_{\kappa \leq \lambda} \{0,1\}^\kappa \), \( |\Pr[X(\lambda, z) = 1] - \Pr[Y(\lambda, z) = 1]| < \lambda^{-c} \).

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

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

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

To define the security of protocol \( \Pi \) 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 \( \Pi \) UC-secure if \( \Pi \) UC-emulates the *ideal protocol* \( \Phi = \text{IDEAL}_\mathcal{F} \). The ideal protocol \( \text{IDEAL}_\mathcal{F} \) 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 \( \mathcal{P} \) 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 \( \text{id} \) its instance gets created with initial internal states. We model this routine by introducing \( \mathcal{F}\text{.INIT}_{\text{id}}() \), which can be called only once. Once an instance of \( \mathcal{F} \) is created within \( \text{id} \), the subsequent calls to \( \text{INIT}_{\text{id}}() \) are ignored. If \( \mathcal{F} \) comes with interface \( \text{INTERFACE} \), the (co)-routine "\( \mathcal{F}\text{.INTERFACE}_{\text{id}}(\text{in}) \)" defines the behavior of the interface for session \( \text{id} \) on input \( \text{in} \), and returns the resulting output, potentially after interacting with the simulator. Every invocation of \( \text{INTERFACE}_{\text{id}} \) may update the internal state of an instance of \( \mathcal{F} \).
**Functionality 1: $\mathcal{F}$-wNIZK**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$T \leftarrow []$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{PROVE}_{\text{ad}}(x, w)$</td>
</tr>
<tr>
<td>3</td>
<td>$\pi \leftarrow S.\text{SIMULATE}_{\text{ad}}(x)$</td>
</tr>
<tr>
<td>4</td>
<td>$T \leftarrow T \cup (x, \pi)$</td>
</tr>
<tr>
<td>5</td>
<td>return $\pi$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{VERIFY}_{\text{ad}}(x, \pi)$</td>
</tr>
<tr>
<td>7</td>
<td>if $(x, \pi) \in T$ then return 1</td>
</tr>
<tr>
<td>8</td>
<td>$w \leftarrow S.\text{EXTRACT}_{\text{ad}}(x, \pi)$</td>
</tr>
<tr>
<td>9</td>
<td>if $(x, w) \in \mathcal{R}$ then $T \leftarrow T \cup (x, \pi)$</td>
</tr>
<tr>
<td>10</td>
<td>if $(w = \text{maul} \land (x, \pi) \in T)$ then $T \leftarrow T \cup (x, \pi)$</td>
</tr>
<tr>
<td>11</td>
<td>if $(x, \pi) \in T$ then</td>
</tr>
<tr>
<td>12</td>
<td>return 1</td>
</tr>
<tr>
<td>13</td>
<td>else</td>
</tr>
<tr>
<td>14</td>
<td>return 0</td>
</tr>
</tbody>
</table>

**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 $\Pi$ is said to UC-emulate $\Phi$ in the presence of $\mathcal{G}$, if for any PPT adversary $A$, there exists a PPT simulator $S$ such that for all PPT environment $\mathcal{Z}$,

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

Here, $\text{EXEC}_{\Pi, \mathcal{G}, \mathcal{A}, \mathcal{Z}}(\lambda, z)$ is defined in terms of $\text{EXEC}_{\mu_{[\Pi, \mathcal{G}]}, \mathcal{A}, \mathcal{Z}}(\lambda, z)$, where the so-called management protocol $\mu$ allows $\Pi$ 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 $\Pi$ UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}$, then the protocol $\rho^\Pi_{\mathcal{G}}$ 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 $\Pi$, 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 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 $G_{GG}$ as in Functionality 2 (similar to, for example, [CNPR22]). As in standard generic type-III bilinear groups, $G_{GG}$ is parameterized by a prime $p$ and three sets $S_i$ for $i \in \{1, 2, t\}$ each of size $p$. $G_{GG}$ starts by initializing three random injections $\tau_i : \mathbb{Z}_p \rightarrow S_i$ for $i \in \{1, 2, t\}$. (This choice can be made efficient in the standard way, via lazy sampling.)

The functionality $G_{GG}$ offers three interfaces to protocols. They can use $G_{GG}$ to access the “canonical” generators $\tau_i(1)$ via CANONICALGen. As with standard generic groups, $G_{GG}$ 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 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, $G_{GG}$ 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, $G_{GG}$ 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 $G_{GG}$ 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 $G_{oGG}$ is defined in Functionality 3. It contains all interfaces of $G_{GG}$, together with two additional ones, OBSERVE and
G-oGG is (implicitly) parameterized with
- A prime number \( p \)
- Sets \( S_1, S_2, S_3 \subseteq \{0,1\}^* \) with \( |S_i| = p \) for all \( i \in \{1,2,3\} \).

G-oGG maintains the following state:
- \( \tau : \mathbb{Z}_p \rightarrow S_i \) three random encoding functions, mapping discrete logs \( x \in \mathbb{Z}_p \) to their randomly encoded group elements \( h \in S_i \).
- \( Var_{i,sid} \) initially empty lists of formal variables. \( \text{//} \) Keeps track of the group \( i \) formal variables belonging to session \( sid \).
- \( R_i[h] \) for \( i \in \{1,2,3\}, h \in S_i \), initially empty sets of polynomials \( \text{//} \) 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 \( Var_{sid} = Var_{1,sid} \cdot Var_{2,sid} : Var_{3,sid} \) 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 \( sid \)).
- \( Legal_{sid} = Var_{sid} \cup Var_{sid} \cdot Var_{sid} \cdot Var_{sid} \) is (implicitly) parameterized with \( \text{G-oGG} \).

\[ \tau : \mathbb{Z}_p \rightarrow S_i \] three random encoding functions, mapping discrete logs \( x \in \mathbb{Z}_p \) to their randomly encoded group elements \( h \in S_i \).

\[ Var_{i,sid} \text{ initially empty lists of formal variables. } \text{//} \text{ keeps track of the group } i \text{ formal variables belonging to session } sid. \]

\[ R_i[h] \text{ for } i \in \{1,2,3\}, h \in S_i \text{, initially empty sets of polynomials } \text{//} \text{ keep track of polynomial representations corresponding to } h \in S_i. \]

\[ Ob \text{ initially empty list of observable actions.} \]

Furthermore, we use the following terms derived from the current state:
- We write \( Var_{sid} = Var_{1,sid} \cdot Var_{2,sid} \cdot Var_{3,sid} \) 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 \( sid \)).
- \( Legal_{sid} = (Var_{sid})^2 \cdot Var_{sid} \cdot Var_{sid} \cdot Var_{sid} \) is the set of polynomials that contain only this session’s variables \( X \in Var_{sid} \), and whose constant term is 0. For example, \( 15X_{sid} + 7Y_{sid} \in Legal_{sid} \) and \( 3X_{sid}Y_{sid} \in Legal_{sid} \), but \( X_{sid} + 3 \not\in Legal_{sid} \) and \( X_{sid} \not\in Legal_{sid} \).

Functionality 3: \( \text{G-oGG} \)

\[ \text{G-oGG} \]

\[ \text{is (implicitly) parameterized with} \]
- \( \text{A prime number } p \)
- \( \text{Sets } S_1, S_2, S_3 \subseteq \{0,1\}^* \text{ with } |S_i| = p \text{ for all } i \in \{1,2,3\}. \]

\[ \text{G-oGG} \]

\[ \text{maintains the following state:} \]
- \( \tau : \mathbb{Z}_p \rightarrow S_i \) three random encoding functions, mapping discrete logs \( x \in \mathbb{Z}_p \) to their randomly encoded group elements \( h \in S_i \).
- \( Var_{i,sid} \) initially empty lists of formal variables. \( \text{//} \text{ keeps track of the group } i \text{ formal variables belonging to session } sid. \)
- \( R_i[h] \) for \( i \in \{1,2,3\}, h \in S_i \), initially empty sets of polynomials \( \text{//} \text{ keep track of polynomial representations corresponding to } h \in S_i. \)
- \( Ob \) initially empty list of observable actions.

Furthermore, we use the following terms derived from the current state:
- We write \( Var_{sid} = Var_{1,sid} \cdot Var_{2,sid} \cdot Var_{3,sid} \) 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 \( sid \)).
- \( Legal_{sid} = (Var_{sid})^2 \cdot Var_{sid} \cdot Var_{sid} \cdot Var_{sid} \) is the set of polynomials that contain only this session’s variables \( X \in Var_{sid} \), and whose constant term is 0. For example, \( 15X_{sid} + 7Y_{sid} \in Legal_{sid} \) and \( 3X_{sid}Y_{sid} \in Legal_{sid} \), but \( X_{sid} + 3 \not\in Legal_{sid} \) and \( X_{sid} \not\in Legal_{sid} \).
Touch. If Observe and Touch are never called, then \( G\cdot\mathsf{oGG} \) behaves identically to \( G\cdot\mathsf{GG} \). In particular, \( G\cdot\mathsf{oGG}.\mathsf{Op}_{\mathsf{sid}}(i, g_1, g_2, a_1, a_2) \) 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 \( G\cdot\mathsf{GG} \) is that the operation additionally keeps track of the way group elements are computed, which we discuss below. Similarly, \( G\cdot\mathsf{oGG}.\mathsf{Pair} \) differs from \( G\cdot\mathsf{GG}.\mathsf{Pair} \) only in maintaining some additional bookkeeping.

Our strategy to restrict observability is similar to (restricted) observable random oracles \([\text{CDG}^+18]\) in that we deploy a form of “domain separation”.\(^{10}\) \( G\cdot\mathsf{oGG} \) 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 \( \mathsf{Touch}_{\mathsf{sid}}(i, g) \) procedure. Reminiscent of the Unix \( \mathsf{touch} \) command, if \( g \) is already in use, nothing happens. Otherwise, \( g \) becomes a generator of the caller’s session \( \mathsf{sid} \). (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 \( \mathsf{Var}_{i, \mathsf{sid}} \) (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 \( R_i[h] \) that explain how the group element has been computed. For a touched generator \( g \) with associated formal variable \( X \), the polynomial representation is simply \( R_i[h] = \{X\} \). The canonical generators are represented with constant polynomials, \( R_i[\tau_i(1)] = \{1\} \).

**Group operations.** When executing group operations, \( G\cdot\mathsf{oGG} \) 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 generators associated with formal variables \( R_i[g_1] = \{X_1\}, R_i[g_2] = \{X_2\} \). If we compute \( “h = 1 \cdot g_1 + 3 \cdot g_2” \), then the corresponding polynomial is \( R_i[h] = \{X_1 + 3X_2\} \). If we further compute \( “h’ = 2 \cdot h + 50 \cdot g_1” \), then \( R_i[h’] = \{52X_1 + 6X_2\} \). Note that by design, polynomials in \( R_1 \) and \( R_2 \) are of degree 1 or 0, and polynomials in \( R_i \) for the target group are of degree at most 2.

It may happen that there are two polynomial representations \( f \neq f’ \) for the same group element \( h \). For this reason, \( R_i[h] \) is formally modeled as a set containing all known representations. However, by Schwartz–Zippel (Lemma 1), for sufficiently large groups, \( R_i[h] \) 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 \( f \in R_i[h] \) associated to each group element \( h \), and sessions to each polynomial variable \( X \in \mathsf{Var}_{\mathsf{sid}} \). This now allows us to establish the observation rules. For this, we say that a group element \( h \) is legal in session \( \mathsf{sid} \) if its associated polynomial(s) \( f \in R_i[h] \) do not contain variables \( X_{\mathsf{sid}’} \in \mathsf{Var}_{\mathsf{sid}’} \) of foreign sessions \( \mathsf{sid}’ \neq \mathsf{sid} \) (and no constant terms, which correspond to the canonical generators). This the set \( \mathsf{Legal}_{\mathsf{sid}} \) in Functionality 3 formally defines the set of legal polynomials for session \( \mathsf{sid} \). 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

\(^{10}\) In Appendix C we give an overview of domain-separation approaches in global observable functionalities.
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 \textsc{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 $R_i[h] = \{4X_{sid} + 3X'_{sid}\}$, where $X_{sid}, X'_{sid} \in \text{Var}_{i,sid}$ are associated with session $sid$, then clearly, $h$ belongs to session $sid$. An $\text{Op}_{sid}$ operation called by a party in session $sid$, resulting in $h$ is an example of an unobservable operation. However, if $R_i[h] = \{4X_{sid} + 3Y_{sid'}\}$, where $Y_{sid'} \in \text{Var}_{i,sid'}$ belongs to session $sid' \neq sid$, then $h$ does not belong to either session. An $\text{Op}_{sid}$ 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 $\text{Pair}_{sid}(h_1, h_2) = h$, we naturally get that if, say, $R_1[h_1] = \{X_{sid}\}$ and $R_2[h_2] = \{3Z_{sid}\}$, then for the result $h$, we get $R_i[h] = \{3X_{sid}Z_{sid}\}$, which indicates that $h$ is legal (unobservable). If, however, instead $R_2[h_2] = \{3Z_{sid'}\}$ with $Z_{sid'} \in \text{Legal}_{2,sid'}$, then the result is illegal (hence observable), since $R_i[h] = \{3X_{sid}Z_{sid'}\} \not\subseteq \text{Legal}_{sid}$.

**Using $G$-\textit{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 \textsc{Touch}ing 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 $f \in R_i[h]$ involves any variable $X \in \text{Var}_{sid}$.

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 $\pi$ 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 \textsc{Touch}ing 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 $g_1, g_2, g_3$, available via \textsc{CanonicalGen} correspond to discrete logarithms $\tau^{-1}_g(g_i) = 1$ and the constant polynomials $R_i[g] = \{1\}$. 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 $\text{Legal}_{sid} = \langle \text{Var}_{sid}\rangle_2 \otimes \text{Var}_{sid}$ corresponding to unobservable polynomials can be written as an ideal).

**Efficiency** Similar to $G$-\textbf{GG}, the observable $G$-\textit{oGG} is also not efficient. In addition to sampling the random encoding functions $\tau$ at the start, the sets $R_i[g]$ in $G$-\textit{oGG} can also blow up to superpoly sizes in the worst case. However, as we argue in Lemma 3, with overwhelming probability, $R_i[g]$ will be a singleton. To make $G$-\textit{oGG} efficient, one can sample $\tau_i$ values lazily (as sketched in Functionality 17 in Appendix F), and if any set $R_i[g]$ 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 $G-oGG$ faithfully models a generic group (as in $G-GG$) with tacked-on observation capabilities. However, an issue of $G-oGG$ when doing security proofs is that the session separation in $G-oGG$ is imperfect. It can happen that some group element belongs to two sessions in $G-oGG$, 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 $G-oSG$, where session separation is perfect by definition and this error event cannot happen. Lemma 3 shows that $G-oGG$ can be securely replaced by $G-oSG$.

In addition to that, $G-oSG$ 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 $X_\alpha, X_\beta$. 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 $a \cdot X_\alpha + b X_\beta$, 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 $X + Y$, 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 $A$ the public key $[X_x, X_y]_2$ and signatures $\sigma_i = [X_{r_i}, X_{s_i}(X_x + m_i X_y)]_1$, so when the adversary outputs a forgery (in the first group), it must be of the form $[\sum \alpha_i X_{r_i} + \sum b_i X_{s_i}(X_x + m_i X_y)]_1$, and hence cannot be forgery” [PS16]. These arguments are inherently symbolic, i.e. in the last step, $X_x, X_y, X_{r_i}$ 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 $G-oSG$ 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 $G-oGG$ replaced by $G-oSG$.
2. Instead of choosing random secrets $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_p^*$, have the UC simulator ask (the extended) $G-oSG$ for corresponding formal variables $X_\alpha, X_\beta \leftarrow \text{GetRnd}()$, and use $\text{COMPUTE}^{\text{SYMBOLIC}}_{\text{sid}}$ to output group elements relative to the secrets. Lemma 4 shows that this switch is undetectable.
3. Have the UC simulator use the algorithm $\text{FINDREP}$ 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 \( G\text{-oSG} \) (+ extensions), which automatically keeps track of the polynomial \( \tau_i^{-1}(h) \) corresponding to each group element \( h \).

The symbolic \( G\text{-oSG} \) 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 \( G\text{-oSG} \) in Functionality 4, which, in contrast to \( G\text{-oGG} \), has perfect separation of sessions. This separation is modeled similarly to \( G\text{-oGG} \), with polynomials. In contrast to \( G\text{-oGG} \), the polynomials will not only be some bookkeeping artifacts \( R \) alongside the actual \( G\text{-GG} \) functionality, but rather the main driver behind group operations. More concretely, the random encoding function \( \tau_i : \mathbb{Z}_p[\text{Var}, \text{SimVar}^\pm] \rightarrow S_i \) 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. \( \text{Op}, \text{Pair} \). In this sense, the polynomial mapping \( \tau_i \) serves two purposes now: It manages the algebraic properties of group elements (managed by \( \tau_i \) over \( \mathbb{Z}_p \) in \( G\text{-oGG} \)) and it is used to decide observability (used to be managed by \( R \) in \( G\text{-oGG} \)).

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

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

**Lemma 3.** Let \( O_{\text{real}} = G\text{-oGG}([\text{CanonicalGen}, \text{Observe}, \text{Touch}, \text{Op}, \text{Pair}]) \). Let \( O_{\text{symb}} = G\text{-oSG}([\text{CanonicalGen}, \text{Observe}, \text{Touch}, \text{Op}, \text{Pair}]) \).

For all algorithms \( B \) that make at most \( q \) oracle queries, it holds that

\[
\Pr[B_{\text{real}} = 1] - \Pr[B_{\text{symb}} = 1] \leq \left( \frac{3q + 1}{2} \right) \cdot 2/p \leq (9q^2 + 3q)/p
\]

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

**Proof (Sketch).** Both \( G\text{-oGG} \) and \( G\text{-oSG} \) use polynomial variables \( X \) to separate sessions and use polynomials (via \( \tau_i^{-1} \) for \( G\text{-oSG} \) and via \( R_i \) for \( G\text{-oGG} \)) to make decisions about observability. The essential difference between \( G\text{-oGG} \) and \( G\text{-oSG} \) is that \( G\text{-oSG} \) (1) keeps track of group elements in terms of variables \( X \) via \( \tau_i : \mathbb{Z}_p[\text{Var}] \rightarrow S_i \) and (2) makes decisions on whether to output a fresh random encoding or an old one w.r.t. the polynomials in \( \tau_i \). \( G\text{-oGG} \), in contrast, (1) keeps track of group elements in terms of random session-separating dlogs \( x \) via \( \tau_i : \mathbb{Z}_p \rightarrow S_i \) and (2) makes decisions on whether to output a fresh random encoding or

---

11 For now, ignore the list \( \text{SimVar} \) of formal variables. It is empty and will only be used in the \( G\text{-oSG} \) extensions (Functionality 5).
**Functionality 4: G-oSG**

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

- \( \tau \) now maps polynomials \((Z_p[\text{Var}, \text{SimVar}^\pm])\) instead of \(Z_p\) to random encodings \(S_i\)
- \( \text{Var}_{sid} \) initially empty lists of polynomial variables
- \( \text{SimVar}_{sid} \) empty lists of polynomial variables \(X_{sid}\). Only used in Functionality 5
- \( \text{SimVal}_{sid} \) empty lists of random scalars \(x_{sid} \in Z_p\) corresponding to \( \text{SimVar}_{sid} \)
- \( \text{Ob}_{sid} \) initially empty list of (globally) observable actions
- \( \text{Ob}_{sid} \) initially empty lists of all actions observable in specific session \(sid\), including actions of parties in session \(sid\) (only read in the G-oSG extensions)
- \( \mathcal{C}_i \) initially empty sets \( \mathcal{C}_i \subseteq S_i \) of group elements that can be the basis for extraction (only read in the G-oSG extensions)

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

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

Function

\[
\text{INIT}(i) \quad /\quad \text{Invoked only upon creation}
\]

1. for \( i \in \{1, 2, 3\} \) do
2. \( \tau_i \leftarrow \{\} \)
3. \( \text{Tau}(i, 1) \)
4. \( \mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{1\} \)

\[
\text{CANONICALGEN}_{sid}(i)
\]

5. return \( \text{Tau}(i, 1) \)

\[
\text{OBSERVE}_{sid}(i)
\]

6. return \( \text{Ob} \)

\[
\text{OP}_{sid}(i, g_1, g_2, a_1, a_2)
\]

7. assert \( (g_1, g_2, a_1, a_2) \in S_i^2 \times Z_p^2 \)
8. for \( j \in \{1, 2\} \) do
9. \( \text{Tau}(i, g_j) \)
10. \( f \leftarrow a_1\tau_j^{-1}(g_1) + a_2\tau_j^{-1}(g_2) \)
11. \( h \leftarrow \text{Tau}(i, f) \)
12. if \( f \notin \text{Legal}_{sid} \) then
13. \( \text{Ob}_{sid} \leftarrow \text{Ob}_{sid} : [(\text{OP}, i, g_1, g_2, a_1, a_2, h)] \)
14. \( \text{Ob}_{sid} \leftarrow \text{Ob}_{sid} : [(\text{OP}, i, g_1, g_2, a_1, a_2, h)] \) for all \( \text{sid'} \) (incl. \( \text{sid} \))
15. \( \text{Ob}_{sid} \leftarrow \text{Ob}_{sid} : [(\text{OP}, i, g_1, g_2, a_1, a_2, h)] \)
16. return \( h \)

\[
\text{TOUT}_{sid}(i, g)
\]

17. if \( g \notin \text{im}(\tau_i) \) then
18. \( \text{Initialize a fresh variable } X \)
19. \( \text{Var}_{sid} \leftarrow \text{Var}_{sid} : [X] \)
20. \( \tau_i(X) \leftarrow g \)
21. \( \mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{g\} \)

\[
\text{TAT}(i, f) \quad /\quad \text{internal}
\]

22. if \( \tau_i(f) \leftarrow f \) then
23. \( \tau_i(f) \leftarrow f \setminus \text{im}(\tau_i) \)
24. return \( \tau_i(f) \)

\[
\text{PAIR}_{sid}(g_1, g_2)
\]

25. assert \( (g_1, g_2) \in S_i \times S_2 \)
26. for \( i \in \{1, 2\} \) do
27. \( \text{TOUT}_{sid}(i, g_i) \)
28. \( f \leftarrow \tau_i^{-1}(g_1) \cdot \tau_i^{-1}(g_2) \)
29. \( h \leftarrow \text{TAT}(i, f) \)
30. if \( f \notin \text{Legal}_{sid} \) then
31. \( \text{Ob}_{sid} \leftarrow \text{Ob}_{sid} : [(\text{PAIR}, i, g_1, g_2, h)] \)
32. \( \text{Ob}_{sid} \leftarrow \text{Ob}_{sid} : [(\text{PAIR}, i, g_1, g_2, h)] \) for all \( \text{sid'} \) (incl. \( \text{sid} \))
33. \( \text{Ob}_{sid} \leftarrow \text{Ob}_{sid} : [(\text{PAIR}, i, g_1, g_2, h)] \)
34. return \( h \)
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.

Functionality 5: G-oSG extensions

For $G$-oSG, computing additional interfaces is straightforward:

\begin{align*}
\text{ComputeAtomic}_{\text{ad}}(i, (h_j, f_j))_{j=1}^n & \\
\text{GetRep}_{\text{ad}}(i, h^*, B) & \\
\text{ComputeSymbolic}_{\text{ad}}(i, (h_j, f_j))_{j=1}^n &
\end{align*}

\begin{align*}
35: & \text{assert } \tau^{-1}_i(h_j) \in \text{Legal}_{\text{ad}} \not\equiv h_j \text{ belongs to session sid and } f_j \in \mathbb{Z}_p[\text{SimVar}_{\text{ad}}] \text{ for all } j \in [n]. \\
36: & h \leftarrow \text{Tau}(i, 0) \not\equiv h = 0 \text{ neutral element} \\
37: & \text{for } j \in [n] \text{ do} \\
38: & \quad a_j \leftarrow f_j(\text{SimVal}_{\text{ad}}) \not\equiv f_j \in \mathbb{Z}_p. \text{ Compute exponent } a_j \text{ from secrets SimVal}_{\text{ad}} \\
39: & \quad h \leftarrow \text{Or}_{\text{ad}}(i, h, h_j, 1, a_j) \not\equiv h \leftarrow h + a_j \cdot h_j \\
40: & \text{return } h \\
\end{align*}

\begin{align*}
45: & \text{Initialize a new variable } X \\
46: & \text{SimVar}_{\text{ad}} \leftarrow \text{SimVar}_{\text{ad}} : \{X\} \\
47: & x \in \mathbb{Z}_p^n \\
48: & \text{SimVal}_{\text{ad}} \leftarrow \text{SimVal}_{\text{ad}} : \{x\} \\
49: & \text{return } X \\
\end{align*}

\begin{align*}
50: & \text{assert } i \in \{1, 2\}, h^* \in \text{im}(\tau_i), B \in (C_i)^n \text{ with } B_j \neq B_k \text{ for } j \neq k. \\
51: & (a_j)_{j=1}^n \leftarrow \text{FindRep}(i, h^*, Ob_{\text{ad}}, B) \\
52: & V = \sum_{j=1}^n a_j \cdot \tau^{-1}_i(B_j) \not\equiv \text{Result as polynomial } V \in \mathbb{Z}_p[\text{Var}_{\text{ad}}] \\
53: & \text{assert } \exists b_j, c_j \in \mathbb{Z}_p[\text{SimVar}_{\text{ad}}] : V = \tau^{-1}_i(h^*) + \text{foreign + missing}, \text{ where foreign} = \sum_{j \not\in \text{Var}_{\text{ad}}} b_j X_j \\
54: & \text{and missing} = \sum_{j \not\in \text{Var}_{\text{ad}}} c_j \cdot \tau^{-1}_i(C_i[j]), \text{ where } C_i[j] \text{ is the } j\text{th element of the set } C_i \text{ according to some canonical ordering}. \\
55: & \text{return } a_1, \ldots, a_n \\
\end{align*}

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

4.2 Extending 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 $G$-oSG UC setting. For this, we extend $G$-oSG with additional interfaces in Functionality 5.

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

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 $\text{Tau}(i, f)$. This functionality heavily uses the fact that the encoding functions $\tau_i$ already work over polynomials. In the original $G\text{-oSg}$, 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 $X_\alpha$ corresponding to secrets of $B$. For example, if $g$ is a generator corresponding to $X_g \in \text{Var}$, and the computation is "$h \leftarrow \alpha^{-2} \cdot g$", then the result $h$ will be internally associated with the polynomial $f = X_\alpha^{-2} \cdot X_g = \tau_i^{-1}(h) \in \mathbb{Z}_p[\text{Var}, \text{SimVar}^\pm]$, 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 $O = G\text{-oSg}[\text{CanonicalGen}, \text{Observe}, \text{Touch}, \text{Op}, \text{Pair}, \text{GetRnd}]$. Let $B^{O, \text{ComputeX}}$ be an algorithm that makes at most $q$ oracle queries. For oracle queries

$$\text{ComputeX}(i, (h_{\ell,j}, f_{\ell,j})_{j=1}^{n_\ell}),$$

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 $B^{O, \text{ComputeSymbolic}}$. If $3q + q' + 1 \leq p$, then

$$\text{Pr} \left[ B^{O, \text{ComputeConcrete}} = 1 \right] - \text{Pr} \left[ B^{O, \text{ComputeAtomic}} = 1 \right] \leq (2q + q') \cdot q'/(p - q).$$

$$\text{Pr} \left[ B^{O, \text{ComputeAtomic}} = 1 \right] - \text{Pr} \left[ B^{O, \text{ComputeSymbolic}} = 1 \right] \leq \left( \frac{3q + 1}{2} \right) \cdot \frac{2d}{(p - 1)}.$$

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

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

Proof. For the first part of the proof, replacing ComputeConcrete with ComputeAtomic cannot be detected by $B$ 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 $q'(p - q)$. $B$ 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 COMPUTE_ATOMIC with COMPUTE_SYMBOLIC cannot be detected unless there is a collision among Laurent polynomials with random input $SimVal$, i.e. two polynomials $f \neq f' \in \text{dom}(\tau_i) \subset \mathbb{Z}_p[\text{Var}, \text{SimVar}]$ such that $f(Val) = f'(Val) \in \mathbb{Z}_p[\text{Var}]$. Note that we are not interested in whether the session-separation variables $\text{Var}$ collide — those remain symbolic in both settings. This is a straightforward application of Lemma 2. Consider any two polynomials $f \neq f' \in \mathbb{Z}_p[\text{Var}, \text{SimVar}]$ queried to $\text{Tau}(i, \cdot)$ for $i \in \{1, 2\}$. By virtue of generic group operations, we can write $f = \sum_j X_j \cdot t_j + t_0$ and $f' = \sum_j X_j \cdot t'_j + t'_0$, where $X_j \in \text{Var}$ and $t_j, t'_j \in \mathbb{Z}_p[\text{SimVar}]$. Because $f \neq f'$, there must be some $t_j \neq t'_j$. From Lemma 2, we know that $\Pr[t_j(SimVal) = t'_j(SimVal)] \leq 2d/(p-1)$. Hence $\Pr[f(SimVal) = f'(SimVal)] \leq 2d/(p-1)$. For polynomials $f = \sum_{j,\ell} X_{j,\ell} \cdot t_{j,\ell} + \sum_j X_j \cdot t_{j,0} + t_0$ belonging to the target group, the same argument holds, i.e. $f \neq f' \Rightarrow \Pr[f(SimVal) = f'(SimVal)] \leq 2d/(p-1)$.

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

### 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 $\xi$). 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 $\tilde{f} = 3X \in \text{Legal}_{sid'}$, 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 $\text{GetRep}$ interface (Functionality 5), defines in Line 53 what we can expect from the algorithm $\text{FindRep}$ given the limited observation information: When extracting a representation for $h^*$, the algorithm $\text{FindRep}$ outputs coefficients that (together with the basis) almost sum up to the polynomial $\tau^{-1}_i(h^*)$. 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 $\text{FindRep}$ as a basis (because $\text{FindRep}$ 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 $\text{FindRep}$ 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 $\text{FindRep}$ works correctly. This is defined in terms of the symbolic computation setting and the interface $\text{GetRep}$, which runs $\text{FindRep}$ with the expected input (in particular with the correct observation list $Ob_{sid}$) and then checks the output.
Function 1: FindRep

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

Lemma 5. Consider \( O = G-oGG, \text{CANONICALGEN, OBSERVE, TOUCH, OP, PAIR, GETRND, COMPUTE_SYMBOLIC, GETREP} \). Let \( B \) be an algorithm that makes at most \( p \) queries. Then
\[
\Pr\left[ B^O \text{ has assertion in Line 53 of Functionality 5 fail} \right] = 0
\]
The proof can be found in Appendix I.

5 UC security of Groth16

![Real world](Real world)
![Ideal world](Ideal world)

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

In Protocol 1, we present the Groth16 protocol \( II-G16 \) in the presence of our global observable generic group functionality \( G-oGG \). The protocol is described in the \( F-CRS \)-hybrid model (Functionality 6). The crucial operation is for-loop starting at Line 1, in which \( F-CRS \) 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 \( F-CRS \) or prover accidentally operates on group elements that are already reserved for another session, which occurs with negligible probability).

Theorem 1. \( II-G16 \) UC-realizes \( F-wNIZK \) in the \( F-CRS \)-hybrid model in the presence of \( G-oGG \). Concretely, for any PPT adversary \( A \), there exists a PPT simulator \( S_{G16} \) such that for every \( Z \) that makes at most \( q_Z \) queries to \( G-oGG \), \( q_p \) queries to the \text{PROVE} interface, and
**Functionality 6: F-CRS**

F-CRS has access to $G$-ogg.

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

$$R_{qp} = \{(a_i)_{i=1}^m, (u_i)_{i=t+1}^m : (\sum_{i=0}^m a_i u_i)(\sum_{i=0}^m a_i v_i) \equiv (\sum_{i=0}^m a_i w_i) \text{ mod } t\}$$

To simplify notation we denote $q_i(\alpha, \beta, x) := \beta u_i(x) + \alpha v_i(x) + w_i(x)$.

F-CRS stores:
- $\sigma$, labels for common reference string
- We use the following compact notation for a vector of encoded group elements with known discrete logs:
- $[x, y, \ldots]_{\text{sid}, i} := \{(G\text{-ogg}.\text{Op}_{\text{sid}}(t, g_{\text{sid}}, i, x), 0), \ldots\}$

**INIT\_sid()** // Invoked only upon creation
1: for $i = 1, 2$ do
2: $g_{\text{sid}, i} \leftarrow S_i$
3: $G\text{-ogg}.\text{Touch}_{\text{sid}}(t, g_{\text{sid}, i})$
4: $x, \alpha, \beta, \sigma, \delta \leftarrow Z_p$
5: $\sigma_1 \leftarrow \{q_i(\alpha, \beta, x)^{-1}, q_i(\alpha, \beta, x)^{-1}\}_{i=1}^m, \{q_i(\alpha, \beta, x)^{-1}\}_{i=1}^m = \{x^t(x^{d-1})\}_{i=1}^m$
6: $\sigma_2 \leftarrow \{\beta, \gamma, \delta, \{x^t(x^{d-1})\}_{i=1}^m\}_{i=2}$
7: $\sigma \leftarrow (\sigma_1, \sigma_2)$
8: return $\sigma$

**Protocol 1: II-G16**

The protocol has access to F-CRS and G-ogg.

**PROVE\_sid(x = \{a_i\}_{i=1}^m, w = \{a_i\}_{i=t+1})**
1: if $(x, w) \notin R_{qp}$ then return $\perp$
2: $\sigma' \leftarrow F\text{-CRS}(G\text{-ogg}, R_{qp})$.GETCRS\_sid()$
3: r, s \leftarrow Z_p$
4: Compute $h \in \mathbb{F}_p^{d-2}[X]$ such that $ht = (\sum_{i=0}^m a_i u_i) - (\sum_{i=0}^m a_i v_i)$
5: $A := [a]_{\text{sid}, 1} \leftarrow \{\sum_{i=0}^m a_i u_i(x) + a + rt\}_{i=1}^m$ // Computed by calling $G\text{-ogg}.\text{Op}_{\text{sid}}$ on $[x^t(x^{d-1})\sigma_{\text{sid}, 1}, [\alpha]_{\text{sid}, 1}, [\beta]_{\text{sid}, 1}]$
6: $B := [b]_{\text{sid}, 2} \leftarrow \{\sum_{i=0}^m a_i v_i(x) + \beta + \delta^t\}_{i=2}$ // Computed by calling $G\text{-ogg}.\text{Op}_{\text{sid}}$ on $[x^t(x^{d-1})\sigma_{\text{sid}, 2}, [\beta]_{\text{sid}, 2}, [\delta]_{\text{sid}, 2}]$
7: $C := [c]_{\text{sid}, 1} \leftarrow \{\sum_{i=0}^m a_i q_i(\alpha, \beta, x)^{-1} + h(x)t(x)^{d-1} + sa + rb – r\delta\}_{i=1}^m$ // Computed by calling $G\text{-ogg}.\text{Op}_{\text{sid}}$ on $[q_i(\alpha, \beta, x)^{d-1}\sigma_{\text{sid}, 1}, [x^t(x^{d-1})\sigma_{\text{sid}, 1}, [\alpha]_{\text{sid}, 1}, [\beta]_{\text{sid}, 1}, [\delta]_{\text{sid}, 1}]$
8: return $(A, B, C)$

**VERIFY\_sid(x = \{a_i\}_{i=1}^m, \pi = (A, B, C))**
9: $\sigma' \leftarrow F\text{-CRS}(G\text{-ogg}, R_{qp}).GETCRS\_sid()$
10: $C_{\text{pub}} \leftarrow \{\sum_{i=0}^m a_i q_i(\alpha, \beta, \gamma)^{-1}\}_{i=1}^m$ // Computed by calling $G\text{-ogg}.\text{Op}_{\text{sid}}$ on $[q_i(\alpha, \beta, x)^{-1}\sigma_{\text{sid}, 1}$
11: return $AB = C_{\text{pub}}[\gamma]_{\text{sid}, 2} + C[\delta]_{\text{sid}, 2} + [\alpha]_{\text{sid}, 1}[\beta]_{\text{sid}, 2}$ // Computed by calling $G\text{-ogg}.\text{Op}_{\text{sid}}$ and $G\text{-ogg}.\text{PAIR}_{\text{sid}}$
queries to the VERIFY interface,

\[
| \Pr[\text{EXEC}_F^{\text{wNIZK}}, Z, S_{G16}, G^{\text{oGG}}(\lambda, z) = 1] - \Pr[\text{EXEC}_{II-G16}, Z, A, G^{\text{oGG}}(\lambda, z) = 1]| \\
\leq 72 \cdot d \cdot (m + d + q_Z + (m + d)q_P + \ell q_V + 1)^2 / (p - 1)
\]

and \( S_{G16} \) performs in total the following operations:

- at most \( 3q_P + 9q_V + 2q_Z + 3d + m + 8 \) queries to \( G^{\text{oGG}} \)
- at most \( (2\ell + 8)q_P + (3q_Z + 2\ell + 2)q_V + (d + 1)(3m + 11) \) field operations

where \( d, m, \ell \) depend on the circuit size (see Functionality 6).

Proof. We first construct a simulator \( S_{G16} \) described in Simulator 1. \( S_{G16} \) consists of two major components: SIMULATE that simulates proof \((A, B, C)\) using \( G^{\text{oGG}}, \text{OP} \) and a secret trapdoor for CRS, and EXTRACT that extracts valid witness upon receiving a statement-proof pair using the OBSERVE interface of \( G^{\text{oGG}} \) and FINDREP (Function 1). Whenever \( Z \) queries \( G^{\text{oGG}} \) in the session with \( \text{sid} \), \( S_{G16} \) forwards its queries to the corresponding wrapper interfaces, and relays back the responses to \( Z \). By simply counting the number of calls to \( G^{\text{oGG}} \) interfaces and local addition, multiplication, and division operations in \( \mathbb{F}_p \) performed by \( S_{G16} \), we obtain the runtime of \( S_{G16} \) stated in the theorem (note that we provide the overall runtime of \( S_{G16} \) taking into account the number of activations through every interface: INIT is called at most once, SIMULATE is called at most \( q_P \) times, EXTRACT is called at most \( q_V \) times, and wrapper interfaces for \( G^{\text{oGG}} \) are called at most \( q_Z \) times, respectively). We define a sequence of hybrids, starting from the ideal run of Groth16 with respect to \( S_{G16}, F^{\text{wNIZK}} \) in the presence of \( G^{\text{oGG}} \) (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 \( S_{G16} \) and \( F^{\text{wNIZK}} \) in the presence of \( G^{\text{oGG}} \). The distribution of the output of \( Z \) in \( H_0 \) is identical to \( \text{EXEC}_{F^{\text{wNIZK}}, Z, S_{G16}, G^{\text{oGG}}} \).

- Hybrid \( H_1 \): Same as \( H_0 \) except that \( G^{\text{oGG}} \) is replaced with its symbolic counterpart \( G^{\text{oSG}} \).

- Hybrid \( H_2 \): Same as \( H_1 \) except that \( F^{\text{wNIZK}} \) is replaced with \( F^{\text{wNIZK}}' \), described in Functionality 20. The difference is that \( F^{\text{wNIZK}}' \) returns the output of the honest verification algorithm as in \( II-G16 \) whenever its VERIFY interface gets invoked, while its PROVE interface remains unchanged.

- Hybrid \( H_3 \): Same as \( H_2 \) except that \( F^{\text{wNIZK}}' \) is replaced with \( F^{\text{wNIZK}}'' \), described in Functionality 21. The difference is that \( F^{\text{wNIZK}}'' \) produces \( \pi = (A, B, C) \) following the honest prover algorithm as in \( II-G16 \) whenever its PROVE interface gets invoked, instead of asking \( S_{G16} \) to simulate \( \pi \).

- Hybrid \( H_4 \): Same as \( H_3 \) except that \( G^{\text{oSG}} \) is replaced with its non-symbolic counterpart \( G^{\text{oGG}} \).

Note that \( H_4 \) is equivalent to the real execution of \( II-G16 \) in \( F^{\text{CRS}} \)-hybrid model in the presence of \( G^{\text{oGG}} \) modulo minor syntactic differences.\(^{12}\)

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 \( G^{\text{oGG}} \) with \( G^{\text{oSG}} \).

\(^{12}\) Concretely, to turn \( H_4 \) into \( \text{EXEC}_{II-G16, Z, A, G^{\text{oGG}}} \) one can apply the following syntactic modifications: (1) CRS generation handled by \( S_{G16} \) is replaced with \( \text{EXEC}_{G^{\text{oGG}}} \), (2) \( F^{\text{wNIZK}}'' \) is viewed as \( II-G16 \), and (3) \( S_{G16} \) is replaced with \( A \). 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 \( G^{\text{oGG}} \).
Now that $H_1, H_2, H_3$ only use $G\text{-oSG}$, the discrete logs of session-specific generators $g_{sid,1}$ and $g_{sid,2}$ are treated as formal variables $X_{sid,1}$ and $X_{sid,2}$, respectively.

- To prove $H_1 \approx H_2$ (Claim 5), we first observe that $Z$ distinguishes $H_1$ and $H_2$ only if the VERIFY interface receives accepting $(x, \pi)$ 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 $F\text{-CRS}$ 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 $S_{G16}$ such that it treats randomness $\alpha, \beta, \ldots$ used for CRS generation and $\mu, \nu$ for proof simulation as formal variables $X_\alpha, X_\beta, \ldots, X_\mu, X_\nu$, and then performs group operations using the $\text{ComputeConcrete}$ extension introduced in Section 4. In the next sub-hybrid, every invocation of $\text{ComputeConcrete}$ is replaced with $\text{ComputeSymbolic}$, 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 $G\text{-oSG}$, there’s a small chance that $Z$ notices such inconsistent “styles” of group operations through the queries to $G\text{-oSG}$. 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 is disclosed between parties. Similar issues arise for public-key encryption and oblivious PRFs [JJK14] when modeled with $G\text{-oGG}$. The security proofs of such schemes fail when using $G\text{-oGG}$, 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 $\text{Observe}_{sid}$ 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 $\text{Op}_{sid'}$. The $\text{Observe}_{sid}$ oracle must include $\text{Op}_{sid'}$ 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 \texttt{Observe}_{\text{sid}} must not observe \texttt{OP}_{\text{sid}} operations on group elements that originated in session \textit{sid}. Otherwise it would obtain leaked information about the EG2 secret key.

The crucial step to escape this conundrum is to observe that \texttt{Observe}_{\text{sid}} is only called by the Groth16 simulator in the \textit{ideal} world. Thus conceptually, we can work with a non-observable generic group (and apply the standard UCGS composition theorem to protocols like II-EG2 in that setting). Only when we want to switch from the concrete protocol II-G16 to the ideal \( F \)-\texttt{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).

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

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

\begin{align*}
\text{Init}, \\
\text{CanonicalGen}_{\text{sid}}, \text{Op}_{\text{sid}}, \text{Pair}_{\text{sid}}, \text{Touch}_{\text{sid}}
\end{align*}
as in $G_{-\text{OGG}}$ (Functionality 3).

\begin{enumerate}
\item \text{Observe}_{\text{sid}}()
  \begin{enumerate}
  \item if $\text{sid} \notin \mathcal{S}$ then \text{\# Restrict caller’s session}
  \item return Ob
  \end{enumerate}
\end{enumerate}

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

\begin{enumerate}
\item \text{Init}_{\text{sid}}, \text{Prove}_{\text{sid}}, \text{Verify}_{\text{sid}}
  \begin{enumerate}
  \item if $\text{sid} \notin \mathcal{S}$ then \text{\# Restrict functionality to sessions $\text{sid} \in \mathcal{S}$}
  \item return $\bot$
  \item continue as in $F_{-\text{WIZK}}$ (Functionality 1).
  \end{enumerate}
\end{enumerate}

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

\begin{enumerate}
\item \text{Init}_{\text{sid}}, \text{Prove}_{\text{sid}}, \text{Verify}_{\text{sid}}
  \begin{enumerate}
  \item if $\text{sid} \notin \mathcal{S}$ then \text{\# Restrict protocol to sessions $\text{sid} \in \mathcal{S}$}
  \item return $\bot$
  \item continue as in II-G16 (Protocol 1).
  \end{enumerate}
\end{enumerate}

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

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

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

Additionally, in order to make sure the Groth16 simulator can call $\text{Observe}_{\text{sid}}$ on the evolved $G_{-\text{OGG}}[\mathcal{S}]$, we need to ensure that the session of any instance of II-G16 (and, consequently, $F_{-\text{WIZK}}$) is one of the allowed sessions $\text{sid} \in \mathcal{S}$. For this, we simply restrict II-G16 and $F_{-\text{WIZK}}$ to work only when instantiated with sessions $\text{sid} \in \mathcal{S}$. We thus consider variants $F_{-\text{WIZK}}[\mathcal{S}]$, II-G16[\mathcal{S}] that restrict $F_{-\text{WIZK}}$, II-G16 to sessions in $\mathcal{S}$. When queried on other sessions they return $\bot$, see Functionalties 8 and 9.13

With these restrictions set up, we show in Fig. 2 how to prove a composition $\rho$ of II-G16 and II-EG2 secure, even though II-G16 requires observability and II-EG2 cannot tolerate observability. The figure depicts that a real system with both II-G16[\mathcal{S}] and II-Enc in the presence of $G_{-\text{OGG}}[\mathcal{S}]$ $\xi$-UC-emulates an ideal system with both $F_{-\text{WIZK}}[\mathcal{S}]$ and $F_{-\text{PKE}}$ in the presence of $G_{-\text{OGG}}[\mathcal{S}]$ for identity bounds $\xi$ that reject all $\text{Observe}_{\text{sid}}$ queries for $\text{sid}$ in $\mathcal{S}$. In more detail, the following steps are taken in the figure:

- **Real to $F$-PKE Hybrid:** We first use the UCGS composition theorem for protocol II-EG2 emulating $F$-PKE, which gives us a system with II-G16[\mathcal{S}] and $F$-PKE in the presence

---

13 This is efficiently implementable. For instance $\mathcal{S}$ could be the set of strings starting with "G16".
of $G_{oGG}[\emptyset]$. This is possible because $G_{oGG}[\emptyset]$ behaves like $G_{GG}$, without observations, which makes $II_{E2G2}$ secure in this setting.

- **F-PKE Hybrid to F-PKE Hybrid with observations**: We switch on observations by replacing $G_{oGG}[\emptyset]$ with $G_{oGG}[\Sigma]$, which is made possible by Lemma 6 (intuitively, this switch cannot be detected because the environment is $\xi$-restricted to not test for $\text{Observe availability}$, as are the protocols. The simulator $S_1$ can be assumed without loss of generality never to call $\text{Observe}$).

- **F-PKE Hybrid with observations to F-PKE, F-wNIZK Hybrid with observations**: We apply the UCGS composition theorem for protocol $II_{G16}[\Sigma]$ UC-emulating $F_{wNIZK}[\Sigma]$ (in the presence of $G_{oGG}[\Sigma]$), which is possible because the simulator $S_2$ is able to ask $G_{oGG}$ for observations.

## 7 Conclusion and future work

In this paper, we have established the restricted observable global generic group functionality $G_{oGG}$ and, as an important application to a widespread SNARK, we have proven Groth16 UC-secure in the $F_{CRS}$ hybrid model in the presence of $G_{oGG}$. We expect the functionality $G_{oGG}$ 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 PIO 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 $G_{oGG}$ 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 using (parts of the) CRS across multiple sessions for NIZK in UC. Our Groth16 works in the $F_{CRS}$-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 $II_{G16}$ session in multiple places is not subroutine respecting).

Second, one could attempt to exchange the local $F_{CRS}$ for a global CRS functionality $G_{CRS}$. 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 \( \tau \)". 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.

Acknowledgment

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.

References


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.

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 \( F \)-NIZK in the (\( F \)-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 \( \Sigma \)-protocols into a NIZK that GUC-realizes \( F \)-NIZK in the (\( G \)-RO, \( F \)-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 \( F \)-NIZK the (\( G \)-RO, \( F \)-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.

FAQ

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

Is there a concrete example of an attack that the UC-AGM is blind to? Suppose we have a protocol \( \rho \) with access to two communication channel functionalities \( F \)-Ch1, \( F \)-Ch2. Now consider the following behavior of a party of \( \rho \):
- \( \rho \) generates a random secret \( sk \overset{\$}{\leftarrow} \mathbb{Z}_p \) and corresponding public key \( pk = g^{sk} \).
- \( \rho \) sends \( pk \) over \( F\text{-Ch1} \) to all other parties.
- If \( \rho \) receives \( pk \) over \( F\text{-Ch2} \), then it reveals its secret key \( sk \).

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

However, in the UC-AGM, this attack is not allowed. This is because the adversary cannot send \( pk \) over \( F\text{-Ch2} \), since it does not know the discrete logarithm of \( pk \) w.r.t. the AGM-basis of \( F\text{-Ch2} \). When \( \rho \) sends \( pk \) over \( F\text{-Ch1} \) to the adversary, the element \( pk \) is added to the AGM-basis of \( F\text{-Ch1} \) (which enables the adversary to send \( pk \) via \( F\text{-Ch1} \) with the trivial representation), but not to the AGM-basis of \( F\text{-Ch2} \). This is a feature because otherwise, sibling functionalities could add group elements to each others’ AGM-bases, which makes certain security proofs impossible. However, it demonstrates how certain attacks are not caught by the UC-AGM.

**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 \( G\text{-oGG} \). 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 (\( F\text{-CRS} \)) is local to each Groth16 (\( \Pi\text{-G16} \)) 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 \( G\text{-oGG} \), 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 \( G\text{-oSG} \), meaning that computing \( g_{sid} - g_{sid} \) does not become observable.

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 \( R_i[h] = \{5 + X_{g_{sid}}\} \not\subseteq \text{Legal}_{sid} \).
**Functionality 10: \( F\)-Setup / \( G\)-Setup**

\[
\text{EVAL}_{\text{eval}}(\vec{x})
\]

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 \( y \)

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 \( \pi \), 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 \( G\)-Setup with some interface \( \text{eval} \), where \( \text{eval} \) may either return random oracle images or the result of generic group operations.

![Real (\( F\)-Setup hybrid) world](image1)

\[ F\text{-Setup} \]

\[ \pi \]

\[ A \]

\[ \text{Environment} \]

\[ \approx \]

\[ \mathcal{S} \]

\[ \mathcal{F}\text{-NIZK} \]

\[ \text{Environment} \]

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

For local functionalities, the situation for observability is very simple: In the real world, \( F\)-Setup can only be accessed by the protocol \( \pi \) and the (local) adversary \( A \). In the ideal world, we can think of \( F\)-Setup as being simulated by \( \mathcal{S} \): if \( A \) in the real world wants to access \( F\)-Setup, then \( \mathcal{S} \) can simply execute this access in its head. \( \mathcal{S} \) may even deviate from the behavior of \( F\)-Setup (e.g., program the random oracle), as long as this is undetectable to the real-world adversary \( A \) and the environment. Regarding observability, this situation is quite comfortable for \( \mathcal{S} \). As all access to (the simulated version of) \( F\)-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 $F$-Setup to the global $G$-Setup, the situation changes. $G$-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 $\pi$) direct access to $G$-Setup (cf. Fig. 4). More specifically, the protocol $\pi$ and the adversary $A$ get to make $G$-Setup queries for their session $sid$ (i.e. calls to $\text{eval}_{sid}$), while the environment gets to make queries for all other sessions $sid' \neq sid$. Note that the environment can indirectly query $\text{eval}_{sid}$ for the protocol’s session $sid$ simply by asking the adversary $A$ to make the query.

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

In contrast to the local $F$-Setup, the simulator $S$ does not fully control $G$-Setup. This makes sense from a composability standpoint: say $G$-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, $S$ does not get to see all accesses to $G$-Setup. While accesses made by the adversary $A$ for session $sid$ in the real world are still conceptually visible to $S$ (given that $S$ can internally simulate $A$), accesses made by the environment for sessions $sid'$ happen without involvement of $S$. This means that the environment can easily circumvent its queries from being observed by $S$. 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 $S$ the ability to observe the queries that the environment makes to $G$-Setup (at least the ones that are relevant to proofs in session $sid$).

Strawman 1: just make all queries observable to everyone. To make the environment’s queries observable, we could just augment $G$-Setup with an interface $\text{Observe}$ that simply outputs the list of all $x$ ever queried to $\text{Eval}$. 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 $G$-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 $G$-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 $G$-Setup queries made by the environment.** As alluded to above, for the NIZKPoK application, the dream scenario would be that

- Our simulator $S$ sees all $G$-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 $G$-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 $G$-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 $G$-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 $G$-Setup to make queries in sessions $sid'$ observable, but queries in session $sid$ unobservable. This would result in, effectively, a parameterized functionality $G$-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 $G$-Setup, if they are proven secure only w.r.t. $G$-Setup[$sid_1$] and $G$-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 $G$-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 $G$-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 $sid$.

As shown in Functionality 11, queries to $G$-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.
Functionality 11: $G$-oSetup

State: $Ob$, an initially empty list of observations.

\[
\text{EVAL}_{\text{sid}}(x) \ // \text{caller's session: sid} \\
1. \text{if } x \text{ does not belong to } \text{sid} \text{ then} \\
2. \quad Ob \leftarrow Ob \cup \{x\} \\
3. \quad y \leftarrow […] \\
4. \quad \text{return } y \\
\]

\[\text{Observe}_{\text{sid}}() \]

5. \text{return } Ob

Functionality 12: $G$-oRO

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

\[
\text{EVAL}_{\text{sid}}(x) \ // \text{caller's session: sid} \\
1. \text{if } x \text{ is not of the form } (\text{sid}, x') \text{ then} \\
2. \quad Ob \leftarrow Ob \cup \{x\} \\
3. \quad T[x] = 1 \text{ then } T[x] \leftarrow S \\
4. \quad \text{return } T[x] \\
\]

\[\text{Observe}_{\text{sid}}() \]

5. \text{return } Ob

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

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

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

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

Note that $S$ does not get to see all queries (which is in contrast to security proofs that assume that the simulator gets full control over $F$-Setup). The simulator $S$ can observe all queries for $x$ that belong to session $\text{sid}$ (this can be checked via Table 1). But the environment can make queries for $x'$ that belong to other sessions $\text{sid}'$. For security proofs of protocols
Claim 5. contains some independent random generator set up multiple generators via generic group model without efficient pairing. Furthermore, instead of allowing sessions to based observation rule of Functionality 3. To simplify this discussion, we concentrate on the $G$ In this section, we discuss earlier attempts at modeling Failed attempts at the $G$-oGG without pairing and Touch. $G$-oGG maintains the following state:
- $\tau: \mathbb{Z}_p \rightarrow S$ a random encoding function
- $R[h]$ for $h \in S$ initially empty sets of polynomials
- $Ob$ initially empty list of observable actions.

\textbf{GetGen}_{sid}() 
1: \textbf{if } $h_{sid} = \perp$ \textbf{then} 
2: \hspace{1em} $h_{sid} \leftarrow S$ 
3: \hspace{1em} Initialize fresh variable $X_{sid}$ 
4: \hspace{1em} $R[h_{sid}] \leftarrow R[h_{sid}] \cup \{X_{sid}\}$ 
5: \hspace{1em} \textbf{return } $h_{sid}$ 

\textbf{Observe}_{sid}() 
6: \hspace{1em} \textbf{return } $Ob$

\textbf{Ob}_{sid}(g_1, g_2, a_1, a_2) 
7: \textbf{assert } (g_1, g_2, a_1, a_2) \in S^2 \times \mathbb{Z}_p^2 
8: \hspace{1em} $h \leftarrow \tau(a_1\tau^{-1}(g_1) + a_2\tau^{-1}(g_2))$ 
9: \hspace{1em} $R[h] \leftarrow R[h] \cup \{R[g_1] + a_1R[g_2] + a_2R[g_2]\}$ 
10: \hspace{1em} \textbf{if } $\exists \in R[h], f \notin \{a \cdot X_{sid} | a \in \mathbb{Z}_p\}$ \textbf{then} 
11: \hspace{1em} $Ob \leftarrow Ob : [(\textbf{Ob}, g_1, g_2, a_1, a_2, h)]$ 
12: \hspace{1em} \textbf{return } $h$

using $G$-oSetup, one must argue that those queries are irrelevant for $S$’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 $H_{sid,-}$, queries to $H_{sid',-}$ 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 $G$-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 $\Pi$ w.r.t. $G$-oSetup. Domain separation basically means that we can compose $\Pi$ with other protocols $\Pi'$ 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 $\Pi'$ can do to our protocol $\Pi$, by making the queries of $\Pi'$ observable. We further limit the damage that adversaries (or simulators) for $\Pi'$ can do, by hiding the (domain-separation-respecting) queries that $\Pi$ makes from them.

D Failed attempts at the $G$-oGG functionality

In this section, we discuss earlier attempts at modeling $G$-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 $R[h]$ 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 $a \cdot X_{sid}$ for some $a \in \mathbb{Z}_p$.

First attempt: Simple $sid + sid = sid$ infection mechanism. Our first attempt (Functionality 14) at $G$-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 $sid$ 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 $\text{Prove}_{sid}(x, w)$, 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 $(a_i)_{i=0}^{\ell+1} = x$ is the public input, $(a_i)_{i=\ell+1} = w$ 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 $g_{\text{intermediate}} = \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid}$ 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 $\text{Verify}_{sid}(x, \pi = (A = h_{sid}, B = h_{sid}, C = g_{\text{intermediate}}))$ on some honest verifier. In the real world, the group operations within $\text{Verify}$ 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 \( C \notin V_{sid} \).\(^{14}\) As a result, the environment can distinguish the two worlds, just using leakage of \( G\text{-oGG} \) operations via \( V_{sid} \) in the Functionality 14 setting.

Overall, Functionality 14 is intuitively unsatisfying because we do not want \( G\text{-oGG} \) 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 VERIFY operations on \( C = g_{intermediate} = \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid} \) will be unobservable in both the real and the ideal world, as the rules of Functionality 13 do not care whether \( g_{intermediate} \) has already been computed by the honest prover or whether the environment (session \( sid^t \)) was the one to compute it for the first time. It just looks at \( g_{intermediate} \) symbolically, using the element’s polynomial representation \( \sum_{i=0}^{m} a_i u_i(x) \cdot X_{sid} \), 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 \( g_{intermediate} \) has been computed by the honest prover. One potential way to fix this is by making sure that when the environment computes \( g_{intermediate} \) for its attack, then \( g_{intermediate} \) is also added to \( V_{sid} \) (at which point the attack fails because \( g_{intermediate} \in V_{sid} \) 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 \( \text{OP}_{sid^t} \) in session \( sid^t \), the result is still added to \( V_{sid} \) when appropriate.

This is formalized in Functionality 15. \( \text{OP} \) now maintains \( V_{sid^t} \) for all \( sid^t \) (including \( sid^t = sid \)), 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 the environment computes \( g_{intermediate} \) in session \( sid^t \), it will also be added to \( V_{sid} \). However, the environment can sidestep this mechanism quite easily: instead of computing \( g_{intermediate} \) as \( \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid} \), it computes it as \( h_{sid} + \sum_{i=0}^{m} a_i u_i(x) \cdot h_{sid} - h_{sid^t} \). The resulting group element is the same, but because the first term \( h_{sid^t} \) is not in \( V_{sid} \), none of the operations/in-termediate results are added to \( V_{sid^t} \). As a result, with a minimal change to the attack above,

---

\(^{14}\) In Groth16, verification happens to only consists of pairing operations and the group operations on \( G_1 \), 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.
is the same, namely $X \sum h \text{leakage exposed by Functionality 15.}$

More concretely, say, the environment wants to compute an element $o$ in $G$ that $g$ to fix this issue is to keep track of session associations more aggressively, i.e. instead of saying the environment adds some other sessions’ generator previous Functionality 15 can be seen as a failure to keep track of group elements. When the environment computes an element $o$ in $G$-oGG, we have learned that the fact whether or not an element is observable again not an issue because the terms $h \sum h \cdot \tau \cdot \tau^{-1}(g_1) + a_2 \tau^{-1}(g_2)$ this operation must be observable. However, the environment can escape observability behavior accordingly.

Third attempt: More aggressively adding elements to sessions. The issue with 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 $g_{intermediate}$ is in $V_{sid}$. Whenever the environment computes $g_{intermediate}$ (in any way), or any element that is loosely 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.

Solution: The polynomial-based observation rule. Reflecting on the failed attempts, we can distill three requirements for a good $G$-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 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 $g_{intermediate}$ 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 $g_{intermediate}$ is the same, namely $X_{sid'} + \sum_{i=0}^{m} a_i u_i(x) \cdot X_{sid} - X_{sid'} = \sum_{i=0}^{m} a_i u_i(x) \cdot X_{sid}$. As a result, Functionality 13 treats the group element $g_{intermediate}$ as part of session $sid$ and exhibits observability behavior accordingly.
**Function 2: FindRep for \( \mathbb{G}_t \)**

1. Parse \( B^{(i)} = (B_1^{(i)}, \ldots, B_n^{(i)}) \)
2. \( \text{Rep}_1[h] \leftarrow 0 \in \mathbb{Z}_p^k \) initially for all \( h \in S_1 \)
3. for \( b \in [n_1] \) do \( \text{Rep}_1[B_1^{(i)}] \leftarrow (\text{Kronecker}_b)^{k_1} \)
4. \( \text{Rep}_2[h] \leftarrow 0 \in \mathbb{Z}_p^{k_2} \) initially for all \( h \in S_2 \)
5. for \( b \in [n_2] \) do \( \text{Rep}_2[B_2^{(i)}] \leftarrow (\text{Kronecker}_b)^{k_2} \)
6. \( \text{Rep}_h[h] \leftarrow 0 \in \mathbb{Z}_p^{n_1+n_2} \) initially for all \( h \in S_1 \)
7. for \( b \in [n_1] \) do \( \text{Rep}_h[B_1^{(i)}] \leftarrow (\text{Kronecker}_b)^{n_1+n_2} \)
8. if \( \text{We write} \) \( \text{Rep}_h[h] = (a_{1i}) \in [n_1], (a_{2i}) \in [n_2] \)
9. for \( ob \in \text{Obsid} \) do //In the order entries appear in \( \text{Obsid} \)
10. if \( ob = \text{Parse} \) then // Group operation
11. \( \text{Rep}_h[h] \leftarrow a_1 \cdot \text{Rep}_h[g_1] + a_2 \cdot \text{Rep}_h[g_2] \)
12. if \( ob = \text{Pair} \) then // Pairing operation
13. \( \text{Rep}_h[h] \leftarrow (0^n, \text{Rep}_h[g_1] \otimes \text{Rep}_h[g_2]) \) //\( c \otimes d := (a \cdot d_b) \in [n_1], c \in [n_2] \)
14. return \( \text{Rep}_h[h] \)

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 \( h_{sid'} + 5 \cdot h_{sid} + h_{sid'} \) and \( 5 \cdot h_{sid} - h_{sid'} \) 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 \( X_{sid} + 5X_{sid} \) 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 \( 5X_{sid} \) and conceptually assigns it to \( V_{sid} \) (as it is not a mix of multiple generators).

### E FindRep compatible with \( \mathbb{G}_t \)

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 \( \mathbb{G}_t \) elements. In Oracle 1, we show the corresponding GetRep oracle for \( \mathbb{G}_t \) elements.

To accommodate pairing operations, both \( \text{GetRep}(t, \cdots) \) as well as \( \text{FindRep}(t, \cdots) \) take bases \( B^{(1)}, B^{(2)}, B^{(4)} \) 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 \( g_1, g_2, g_{3} \) are basis elements, we want to be able to output coefficients \( a_1, a_{1,1} \) such that \( h^* = a_1 g_t + a_{1,1} e(g_1, g_2) \) (modulo some terms).
First, note that $G\cdot F$ Proof of Lemma 3 as basis elements into multiplying the known representation vector entries (mirroring $Rep_{out}$ behaves like $\tau$) in Functionality 17. However, the intermediate uniform preimage $f$ in Line 31, but falls back to $x'$ lazily $\tau^{-1}(h)$ which are, in a sense, “missing” basis elements). However, all other terms get correct coefficients from $FindRep$, where “correct” means consistent to $\tau_i(h)$. In particular, if some $G_t$ 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.

Oracle 1: $G\cdot oSG$ extension: $GetRep$ for $G_t$

<table>
<thead>
<tr>
<th>GetRep$_\text{aselt}(h^*, B^{(1)}, B^{(2)}, B^{(3)}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: assert $h^*$ is $\text{im}(\tau_i)$ and $B^{(i)} \subseteq C_i$ for $i \in {1, 2, t}$</td>
</tr>
<tr>
<td>2: Parse $B^{(i)} = (B^{(i)}<em>1, \ldots, B^{(i)}</em>{k_i})$</td>
</tr>
<tr>
<td>3: $(\langle a_i \rangle)<em>{i \in [n_i]; \langle a_i, e \rangle</em>{i \in [n_i], \ell \in [\ell_i]} \rightarrow FindRep(h^*, Ob, Ob_{sid}, B^{(1)}, B^{(2)}, B^{(3)})$</td>
</tr>
<tr>
<td>4: assert $\sum_{j \in [m_i]} a_j \cdot \tau_i^{-1}(B^{(i)}<em>j)$ + $\sum</em>{t \in [t_i]} \sum_{j \in [m_i]} a_{j, t} \cdot \tau_i^{-1}(B^{(j)}_t) \cdot \tau_j^{-1}(B^{(j)}<em>t) = \tau_i^{-1}(h^*)$ mod $\langle Var</em>{sid}, \tau_i^{-1}(C_i \setminus B^{(i)}_1), \tau_j^{-1}(C_j \setminus B^{(j)}<em>t) \rangle</em>{\text{Z}_p[\text{Var}, \text{SimVar}]$</td>
</tr>
<tr>
<td>5: return $(\langle a_i \rangle)<em>{i \in [n_i]; \langle a_i, e \rangle</em>{i \in [n_i], \ell \in [\ell_i]}$</td>
</tr>
</tbody>
</table>

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 $\sum e(\sum y_k, \sum z_\ell) = \sum e(y_k, z_\ell)$).

Similar to the proof of Lemma 5 (see Appendix I), we can also show that whenever $FindRep$ sets some $Rep_{[h]}$ value, then that value is a good representation of the element $\tau_i^{-1}(h)$ modulo $\langle Var_{sid}, \tau_i^{-1}(C_i \setminus B^{(1)}_1), \tau_j^{-1}(C_j \setminus B^{(j)}_t) \rangle_{\text{Z}_p[\text{Var}, \text{SimVar}]$. 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 $\tau_i(h)$. In particular, if some $G_t$ 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 $G\cdot oGG$ samples all group element encodings in the beginning ($\tau_i \overset{\$}{\leftarrow} \text{Inj}(\text{Z}_p, S_i)$). In contrast, $G\cdot oSG$ lazily samples the values for $\tau_i$ (this is because there are no injective functions from $\text{Z}_p[\text{Var}, \text{SimVar}^{\pm 1}]$ to $S_i$). To make $G\cdot oGG$ more like $G\cdot oSG$ in that regard, consider the lazy sampling version of $G\cdot oGG$ in Functionality 17: Whenever $\tau_i(x)$ is first accessed ($x \notin \text{dom}(\tau_i)$), a random unused image $h \overset{\$}{\leftarrow} S_i \setminus \text{im}(\tau_i)$ is chosen for $x$. Whenever $\tau_i^{-1}(h)$ is first accessed ($h \notin \text{im}(\tau_i)$), a random unused preimage $x \overset{\$}{\leftarrow} \text{Z}_p \setminus \text{dom}(\tau_i)$ is chosen for $h$. We push the code for first-access of $\tau_i$ into a new internal interface $\text{TAU}_{sid}$. We push the code for first-access of $\tau_i^{-1}$ into $\text{TOUCH}_{sid}$ because it precedes all accesses of $\tau_i^{-1}$. Of course, this version is perfectly indistinguishable from the original $G\cdot oGG$ (this is easy to see because the $\tau_i : \text{Z}_p \rightarrow S_i$ are bijections).

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

Note that the lazy sampling of $\tau_i'$ preimages $x'$ in $\text{TOUCH}$ first tries to use some uniform $x' \overset{\$}{\leftarrow} \text{Z}_p$ in Line 31, but falls back to $x' \overset{\$}{\leftarrow} \text{Z}_p \setminus \text{dom}(\tau_i')$ (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 $G\cdot oHG\cdot 0$ and $G\cdot oHG\cdot 1$ unless during execution of $G\cdot oHG\cdot 0$, we run into two polynomials $f \neq f' \in \text{P}$ that collide, i.e. $f(Val) = f'(Val)$. Schwartz-Zippel (Lemma 1) establishes that such collisions are unlikely.
Functionality 17: $\mathcal{G}$-oGG with lazily sampled $\tau$

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

**Invariant 1.** Before and after any invocation of the oracles, it holds that for all $i, a \in \{1, 2\}$

$\exists \tau_i(a) \iff \exists \tau_i'(a)$

By design of $\mathcal{G}$-oGG, $\tau_i(a)$ and $\tau_i'(a)$ behave exactly the same. This implies that if $\tau_i(a)$ does not occur, then $\tau_i'(a)$ does not occur, and vice versa.

Proof (Lemma 3). Consider the event “coll” that at some point during execution of $\mathcal{G}$-oHG-0, there are $f, f' \in \mathcal{P}$ such that $f \neq f'$ but $f(\text{Val}) = f'(\text{Val})$. Because polynomials in $\mathcal{P}$ are at most of degree 2, the values in $\text{Val}$ are chosen uniformly and independently at random, and the set $\mathcal{P}$ of polynomials does not depend on $\text{Val}$ (all responses of $\mathcal{G}$-oHG-0 to $B$ are independent of $\text{Val}$), we can apply Lemma 1 pairwise to all $f, f'$. This gives us that $\Pr[f(\text{Val}) = f'(\text{Val}) = 0] \leq \deg(f - f')/p \leq 2/p$ for all $\{f, f'\} \in \binom{\mathcal{P}}{2}$. Every query to $B$'s oracles adds at most three new polynomials to $\mathcal{P}$, in addition to the initial entry $1 \in \mathcal{P}$. With union bound over all $\binom{\binom{\mathcal{P}}{2}}{2}$ pairs, we get

$$\Pr[\text{coll}] = \Pr[\exists f \neq f' \in \mathcal{P} : f(\text{Val}) = f'(\text{Val})] \leq \left(\frac{3q + 1}{2}\right) \cdot 2/p.$$
Functionality 18: $G$-oSG

We write

$$\tau \sim \tau' \sim Ob \sim \text{Legal}$$

if

$$\text{assert } \tau_1 : \tau_2 \sim \tau_3 \sim \tau_4 \sim \text{Ob} \sim \text{Legal}$$

then return

$$\tau$$

We write

$$Var_{\text{sid}}, Var$$ as before. We analogously define $Val_{\text{sid}}$ and $Val$.

$$\text{Legal}_{\text{sid}} = (\text{Var}_{\text{sid}}) \cup \text{Var}_{\text{sid}}$$ / As in $G$-oSG and $G$-oSG (the formulation in the latter is equivalent since SimVar_{\text{sid}} = (\text{sid}) is empty in the context of Lemma 3)

| Functionality 18: $G$-oSG
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- $\tau_1 : Z_p[\text{Var}] \rightarrow S_1$ // as in $G$-oSG</td>
</tr>
<tr>
<td>- $\tau'_1 : Z_p \rightarrow S_1$ // as in $G$-oSG</td>
</tr>
<tr>
<td>- $P$ set of polynomials seen during the game // bookkeeping for proof</td>
</tr>
<tr>
<td>- $\text{R}_{\text{sid}}$ initially empty map from representations to sets of polynomials // as in $G$-oSG</td>
</tr>
<tr>
<td>- $\text{Var}_{\text{sid}}$ initially empty lists of polynomial variables // as in $G$-oSG</td>
</tr>
<tr>
<td>- $\text{Val}<em>{\text{sid}}$ initially empty lists of random values corresponding to the variables of $\text{Var}</em>{\text{sid}}$ // Uniform version (with potential collisions) of $\text{Val}'$. Used for Schwartz-Zippel</td>
</tr>
<tr>
<td>- $\text{Val}'<em>{\text{sid}}$ initially empty lists of random values corresponding to the variables of $\text{Var}</em>{\text{sid}}$ // Version that potentially deviates from $\text{Val}$ because $\text{Val}'$ does not contain duplicates. Used to bridge $G$-oSG into $G$-oSG in the proof, where polynomials $f$ in $G$-oSG correspond to scalars $f(\text{Val}')$ in $G$-oSG</td>
</tr>
<tr>
<td>- $\text{Ob}$ initially empty list of (globally) observable actions // as in $G$-oSG, $G$-oSG</td>
</tr>
</tbody>
</table>

We write

$$\text{We write } Var_{\text{sid}}, Var$$ as before. We analogously define $Val_{\text{sid}}$ and $Val$.

$$\text{Legal}_{\text{sid}} = (\text{Var}_{\text{sid}}) \cup \text{Var}_{\text{sid}}$$ / As in $G$-oSG and $G$-oSG (the formulation in the latter is equivalent since SimVar_{\text{sid}} = (\text{sid}) is empty in the context of Lemma 3)

<table>
<thead>
<tr>
<th>INIT() // Invoked only upon creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for $i \in {1, 2, t}$ do</td>
</tr>
<tr>
<td>2. $\tau_i \gets {}$</td>
</tr>
<tr>
<td>3. $\tau'_i \gets {}$</td>
</tr>
<tr>
<td>4. $g_i \gets \text{TAU}(i, 1, 1)$</td>
</tr>
<tr>
<td>5. $R[g_i] \gets {1}$</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>CANONICALGEN_{\text{sid}}(\text{Var})</td>
</tr>
<tr>
<td>6. return $\text{TAU}(i, 1, 1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{TAU}(i, f, a)$ // internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>7. $P \gets P \cup {f}$</td>
</tr>
<tr>
<td>8. if $b = 0 \wedge \tau_i(f) = \bot \vee b = 1 \wedge \tau_i'(a) = \bot$ then</td>
</tr>
<tr>
<td>9. if $b = 0$ then $h \leftarrow S_i \setminus \text{im}(\tau_i)$</td>
</tr>
<tr>
<td>10. if $b = 1$ then $h \leftarrow S_i \setminus \text{im}(\tau_i')$</td>
</tr>
<tr>
<td>11. $\tau_i(f) \leftarrow h$</td>
</tr>
<tr>
<td>12. $\tau_i'(a) \leftarrow h$</td>
</tr>
<tr>
<td>13. if $b = 0$ then return $\tau_i(f)$</td>
</tr>
<tr>
<td>14. if $b = 1$ then return $\tau_i'(a)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{Ob}_{\text{sid}}(i, g_1, g_2, a_1, a_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>15. assert $(g_1, g_2, a_1, a_2) \in S_1^2 \times Z_p^2$</td>
</tr>
<tr>
<td>16. for $j \in {1, 2}$ do</td>
</tr>
<tr>
<td>17. $\text{Touched}(i, g_1)$</td>
</tr>
<tr>
<td>18. $f \leftarrow a_1 \tau_{i-1}(g_1) + a_2 \tau_{i-1}(g_2)$</td>
</tr>
<tr>
<td>19. $a \leftarrow a_1 \tau_{i-1}(g_1) + a_2 \tau_{i-1}(g_2)$</td>
</tr>
<tr>
<td>20. $h \leftarrow \text{TAU}(i, f, a)$</td>
</tr>
<tr>
<td>21. $R[h] \leftarrow R[h] \cup (a_1 R_1[g_1] + a_2 R_2[g_2])$</td>
</tr>
<tr>
<td>22. if $b = 0 \wedge f \notin \text{Legal}_{\text{sid}}$ then</td>
</tr>
<tr>
<td>23. $\text{Ob} \leftarrow \text{Ob} : ([\text{Ob}, i, g_1, g_2, a_1, a_2, h])$</td>
</tr>
<tr>
<td>24. if $b = 1 \wedge R[h] \notin \text{Legal}_{\text{sid}}$ then</td>
</tr>
<tr>
<td>25. $\text{Ob} \leftarrow \text{Ob} : ([\text{Ob}, i, g_1, g_2, a_1, a_2, h])$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TOUCH_{\text{sid}}(i, g, a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26. if $b = 0 \wedge g \notin \text{im}(\tau_i) \vee b = 1 \wedge R_i[g] = \emptyset$ then</td>
</tr>
<tr>
<td>27. Initialize a fresh variable $X$</td>
</tr>
<tr>
<td>28. $\text{Var}<em>{\text{sid}} \leftarrow \text{Var}</em>{\text{sid}} : [X]$</td>
</tr>
<tr>
<td>29. $\tau_i(X) \leftarrow g$</td>
</tr>
<tr>
<td>30. $R_i[g] \leftarrow {X}$</td>
</tr>
<tr>
<td>31. $x \leftarrow Z_p$</td>
</tr>
<tr>
<td>32. if $\tau_i'(x) \neq \bot$ then</td>
</tr>
<tr>
<td>33. $x' \leftarrow Z_p \setminus \text{dom}(\tau_i')$</td>
</tr>
<tr>
<td>34. else</td>
</tr>
<tr>
<td>35. $x' \leftarrow x$</td>
</tr>
<tr>
<td>36. $\text{Var}<em>{\text{sid}} \leftarrow \text{Var}</em>{\text{sid}} : [x]$</td>
</tr>
<tr>
<td>37. $\text{Var}<em>{\text{sid}}' \leftarrow \text{Var}</em>{\text{sid}}' : [x']$</td>
</tr>
<tr>
<td>38. $\tau_i(x') \leftarrow g$</td>
</tr>
<tr>
<td>39. $P \leftarrow P \cup {X}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBSERVE_{\text{sid}}()</th>
</tr>
</thead>
<tbody>
<tr>
<td>40. return $\text{Ob}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAIR_{\text{sid}}(g_1, g_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>41. assert $(g_1, g_2) \in S_1 \times S_2$</td>
</tr>
<tr>
<td>42. for $i \in {1, 2}$ do</td>
</tr>
<tr>
<td>43. $\text{Touched}(i, g_1)$</td>
</tr>
<tr>
<td>44. $f \leftarrow \tau_{i-1}^{-1}(g_1) \cdot \tau_{i-1}^{-1}(g_2)$</td>
</tr>
<tr>
<td>45. $a \leftarrow \tau_{i-1}^{-1}(g_1) \cdot \tau_{i-1}^{-1}(g_2)$</td>
</tr>
<tr>
<td>46. $h \leftarrow \text{TAU}(i, f, a)$</td>
</tr>
<tr>
<td>47. $R[h] \leftarrow R[h] \cup (R_1[g_1] \cdot R_2[g_2])$</td>
</tr>
<tr>
<td>48. if $b = 0 \wedge f \notin \text{Legal}_{\text{sid}}$ then</td>
</tr>
<tr>
<td>49. $\text{Ob} \leftarrow \text{Ob} : ([\text{PAIR}, i, g_1, g_2, h])$</td>
</tr>
<tr>
<td>50. if $b = 1 \wedge R_i[h] \notin \text{Legal}_{\text{sid}}$ then</td>
</tr>
<tr>
<td>51. $\text{Ob} \leftarrow \text{Ob} : ([\text{PAIR}, i, g_1, g_2, h])$</td>
</tr>
</tbody>
</table>

return $h$
\[ i \in \{1, 2, t\}, h \in S, \]
\[ T_i(h) = (h, \bot, \emptyset, \emptyset) \]
\[ \text{or } T_i(h) = (h, f, \{f(Val')\}, \{f\}) \quad \text{for some } f \in P \]

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

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 \(\tau_i\), the concrete discrete logarithms of \(\tau_i\), and the \(G\)-oGG bookkeeping polynomials \(R_i\). Namely, \(R_i[h]\) is essentially just \(\tau_i^{-1}(h)\), and we get from \(\tau_i^{-1}(h) \in \mathbb{Z}_p[\text{Var}]\) 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 \(Val = Val'\). This invariant claims, essentially, that there is no disparity between the uniformly chosen preimages \(Val \xrightarrow{\$} \mathbb{Z}_p\) and the (collisionless) preimages \(Val'\) used for \(\tau_i\).

**Corollaries of invariants 1,2.** As a consequence of the first invariant, we immediately get that \(\text{im}(\tau_i) = \text{im}(\tau_i)\) (given that any \(h\) is either in the image of none of the \(\tau_i, \tau_i\) functions, or in both of them). Furthermore, because \(\tau_i, \tau_i\) are functions, any \(f \in \mathbb{Z}_p\) 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 \(-\text{coll}, \text{invariant 1 and invariant 2 both hold before and after any oracle call to } G\)-oGG-o.

**Proof:** After Init. The first invariant is clearly fulfilled in the beginning, after INIT, where the \(T_i\) contain the canonical generator entries \((g_1, 1, \{1\}, \{1\})\), and the entries for all other \(h \neq g_1\) are of the empty kind. The second invariant is trivially fulfilled in the beginning, given that \(Val = Val' = ()\) is empty.

In the following, we assume that the invariants hold before any oracle call, and prove that they are preserved, assuming \(-\text{coll}.

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

The invariant is also preserved by queries to \(\text{OP}_{\text{sid}}(i, g_1, g_2, a_1, a_2)\) and \(\text{PAIR}_{\text{sid}}(g_1, g_2)\): For both interfaces, the invariant holds after the internal call to \(\text{TOUCH}\) (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 \(f(Val') = a\). Hence \(\text{TAU}_{\text{sid}}(i, f, a)\) is called with \(a = f(Val')\) in Line 20 and 46, which fulfills the “for all input \((i, f, a)\) ever supplied to TAU, it holds that \(f(Val') = a\)” 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 \(\text{TAU}_{\text{sid}}(i, f, f(Val'))\) sets the previously empty entry \(T(h)\) to \((h, f, \{f(Val')\}, \cdot, \cdot)\).

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 \((h_2, f_2, \{a\}, \cdot)\) containing \(a\) before executing TAU. Then \(a = f_2(Val')\) by invariant 1. Hence \(f(Val') = f'(Val') = a\). By invariant 2, \(f(Val) = f'(Val)\), and by injectivity of \(\tau_i\), \(f \neq f_2\), i.e. we get a collision between \(f(Val)\) and \(f_2(Val)\). Overall, Line 12 does not overwrite any prior \(\tau_i'(a) = h_2\) value unless the event coll happens.

After the internal call to TAU, because \(R_i[g]\) = \(\{\tau_i^{-1}(g)\}\) for the inputs \(g_1, g_2\) by invariant 1, Op/Pair sets \(R[h]\) to \(\{f\}\) in Line 21 and 47, which gives us the entry \(T_i(h)\) of the form \((h, f, \{f(Val')\}, \{f\})\), preserving invariant 1, as required.

**Preservation of Val' 2.** The second invariant is preserved by \(\text{TOUCH}\) assuming \(-\text{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 \( T(h) = (h, f, \{ x \}) \) entry containing \( x \). By invariant 1, \( T(h) \) is well-formed, i.e. \( f(Val') = x = X(Val', x) \). Furthermore, because invariant 2 holds before \textsc{Touch}, we have \( Val' = Val \), hence \( f(Val') = f(Val) = f(Val, x) \) (the last equality holds because \( f \) does not contain \( X \)). In this scenario, \textsc{Touch} would proceed to compute the new \( Val ← Val : [x] \), and \( X \) is added to \( P \). This means that after \textsc{Touch}, there is a collision between \( f \neq X \) (inequality because \( X \) is a fresh variable) with \( f(Val) = X(Val) = x \), which violates the assumption that \( ¬\text{coll} \). 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 \textsc{Touch} is called internally). 

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

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

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

- Regarding Line 8: no difference. By invariant 1, \textsc{Ta} is called with input of the form \( (i, f, a = f(Val')) \). Consider two cases: (1) if \( \tau_i(f) \neq ⊥ \), then by invariant 1, there is an entry \( T_i(h) = (h, f, \{ a \}, \cdot) \), hence \( \tau_i(a) \neq ⊥ \). (2) If \( \tau_i'(a) \neq ⊥ \), then also \( \tau_i(f) \neq ⊥ \). In this case, there is already an entry \( T_i(h_2) = (h_2, f_2, \{ a \}, \cdot) \) containing \( a \). Assume, for contradiction, that \( \tau_i(f) = ⊥ \). Then \textsc{Ta} would reassign \( \tau_i'(a) \) to \( h \neq h_2 \), making the entry \( T_i(h_2) = (h_2, f_2, \{ a \}, \cdot) \) violate invariant 1 after the update. The invariant cannot be violated, hence necessarily, \( im(f') \neq ⊥ \). Overall, either both \( \tau_i(f) \) and \( \tau_i'(a) \) are \( ⊥ \), or neither.

- Regarding Line 9 and 10: no difference because \( im(\tau_i) = im(\tau_i') \) (as observed in the invariant corollaries above)

- Regarding Line 13 and 14: no difference. By invariant 1, \textsc{Ta} is called with input of the form \( (i, f, a = f(Val')) \). For \( b = 0 \), using invariant 1, \textsc{Ta} returns the unique \( h \) such that \( T(h) = (h, f, a, \cdot) \). For \( b = 1 \), \textsc{Ta} must return the same \( h \) because \( a \) only appears once in \( T \) (see invariant corollaries).

- Regarding Line 22, 24, 48, 50: no difference because \( R_i[h] = \{ f \} \) by invariant 1.

- Regarding Line 26: no difference because \( im(\tau_i) = \{ g \mid R_i[g] \neq ⊥ \} \) 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 \textsc{ComputeConcrete} and \textsc{ComputeAtomic}

The difference between \textsc{ComputeConcrete} and \textsc{ComputeAtomic} is that the former calls \textsc{Ta} on all intermediate results (partial sums) \( t_j ← \sum_{\ell=1}^j \tau_i^{-1}(h_\ell) \cdot f_\ell(\text{SimVal}_{sid}) \), whereas the latter only calls \textsc{Ta} on the final result. Functionality 19 encapsulates this essential difference formally: \textsc{Compute} (corresponding to \textsc{ComputeConcrete} / \textsc{ComputeAtomic}) calls a procedure \textsc{PhantomTa}(i, t_j) for the intermediate results \( t_j \), which allocates a random encoding \( h_j = \tau_i(t_j) \) 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 \textsc{Ta}(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 \( B \) has not seen the encodings.

Functionality 19 perfectly emulates both \((\mathcal{O}, \text{ComputeConcrete})\) and \((\mathcal{O}, \text{ComputeAtomic})\), unless it aborts.\(^{15}\) This happens if \( B \) 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 \( Q = Q_1 \cup Q_2 \cup Q_t \).

- There are at most \( q' \geq |Q| \) elements in \( Q \) because elements are added only for intermediate Compute results.
- \( B \) 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 \( |Q|/(p-q) \leq q'/(p-q) \).
- \( B \) can make a guess by querying some \( h \) to Compute or Touch. Hence \( B \) 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 \( B \) makes Functionality 19 abort with probability at most \( (2q + q') \cdot q'/(p-q) \).

\(^{15}\) We are using here that \( B \) 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 35 of ComputeConcrete, none of the Op calls are (globally) observable.
Overall, with the difference lemma, we get that
\[
\left| \Pr[B^G_{\text{ComputeConcrete}} = 1] - \Pr[B^G_{\text{ComputeAtomic}} = 1] \right| \leq \Pr[\text{abort}] \leq (2q + q') \cdot q'/(p - q)
\]
as required.

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

The next lemma (Lemma 7) alleviates a technical concern. Whenever the UC simulator $S$ makes a group operation query, the result of that query gets assigned a random image in $\tau_i$. As a consequence, in $G\text{-oSG}$, it is no longer true that it does not matter how $S$ computes a group element 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 $g \in S_i$ and querying it to $\text{TOUCH}_{ad}(i, g)$. This $\text{TOUCH}$ 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 $A^{G\text{-oSG},B_i}, B_0^{\text{ComputeConcrete}}, B_1^{\text{ComputeConcrete}}$ be algorithms such that both $B_i$ just output the list of their $h_j \leftarrow \text{ComputeConcrete}(i_j, \cdot)$ query results in the format $((i_1, h_1), (i_2, h_2), \ldots)$. We say that $B_0, B_1$ are perfectly equivalent (w.r.t. $A$) if for any invocation $B(x)$ that $A$ makes, $\Pr[y = (\tau_i^{-1}(h_j))_{j=1}^n \mid (i_j, h_j)_{j=1}^n \leftarrow B_0(x)] = \Pr[y = (\tau_i^{-1}(h_j))_{j=1}^n \mid (i_j, h_j)_{j=1}^n \leftarrow B_1(x)]$ for all $y$.

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

If $B_0, B_1$ are perfectly equivalent (with respect to $A$) and $3q + q' + 1 \leq p$, then
\[
\left| \Pr[A^{G\text{-oSG},B_0^{\text{ComputeConcrete}}} = 1] - \Pr[A^{G\text{-oSG},B_1^{\text{ComputeConcrete}}} = 1] \right| \leq 2 \cdot (2q + q') \cdot q'/(p - q)
\]

**Proof.** From Lemma 4, we know that
\[
\left| \Pr[A^{G\text{-oSG},B_0^{\text{ComputeConcrete}}} = 1] - \Pr[A^{G\text{-oSG},B_1^{\text{ComputeAtomic}}} = 1] \right| \leq (2q + q') \cdot q'/(p - q).
\]

Analyzing $B_0^{\text{ComputeAtomic}}$, notice that querying $\text{ComputeAtomic}$ with result $h$ amounts to just querying $h \leftarrow \text{TAU}(i, f)$, i.e. there are no other side effects. Since the inputs $(i, f)$ are distributed the same between $B_0$ and $B_1$ by the lemma’s prerequisites, one can conclude that there is no difference between $B_0$ and $B_1$ in this setting. This means that
\[
\left| \Pr[A^{G\text{-oSG},B_0^{\text{ComputeAtomic}}} = 1] - \Pr[A^{G\text{-oSG},B_1^{\text{ComputeAtomic}}} = 1] \right| = 0.
\]

Overall,
\[
\left| \Pr[A^{G\text{-oSG},B_0^{\text{ComputeConcrete}}} = 1] - \Pr[A^{G\text{-oSG},B_1^{\text{ComputeConcrete}}} = 1] \right|
\leq \left| \Pr[A^{G\text{-oSG},B_0^{\text{ComputeAtomic}}} = 1] - \Pr[A^{G\text{-oSG},B_1^{\text{ComputeAtomic}}} = 1] \right|
+ 2 \cdot (2q + q') \cdot q'/(p - q)
= 0 + 2 \cdot (2q + q') \cdot q'/(p - q)
\]

52
I  Proof of Lemma 5

Proof (Lemma 5). For a representation $Rep = (a_j)_{j=1}^n$, we write the corresponding polynomial $V(Rep) = \sum_{j=1}^n a_j \cdot \tau_i^{-1}(B_j) \in \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm}]$. Let $V(h^*) = (a_j^*)_{j=1}^n$ be the result of a FINDREP$(i, h^*, \mathsf{Ob}_{sid}, B)$ call, and let $V(h^*)$ be its corresponding polynomial. Let $l = \langle \mathsf{Var}_{sid} \cdot \tau_i^{-1}(C_i \setminus B) \rangle \mathbb{Z}_p[\mathsf{Var}, \mathsf{SimVar}^{\pm}]$ be the ideal of, loosely speaking, unobservable polynomials, which is formed by variables of other sessions $sid' \neq sid$ and generators $g \in C_i$ not supplied as basis input via $B$. The assertion in Line 53 is equivalent to checking that $V(Rep[h^*]) = \tau_i^{-1}(h^*) \mod l$.

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

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

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

It remains to argue that indeed, $V(Rep[g_j]) = \tau_i^{-1}(g_j)$ ($j \in \{1, 2\}$). For this, consider the following case distinction about how $g_j$ first appeared in $G$-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 $l$, we have $\tau_i^{-1}(g_j) = 0 \mod l$.

- $g_j$ was first queried to $\mathsf{Touch}_{sid'}$ (either directly or indirectly via $\mathsf{Op}, \mathsf{PAIR}$). 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 $l$, we have $\tau_i^{-1}(g_j) = 0 \mod l$ as required.

- $g_j$ was first seen as the result of a $\mathsf{ComputeSymbolic}_{sid'}$ query. Similarly to $\mathsf{Touch}$,
  - 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 $\tau_i^{-1}(g_j) = 0 \mod l$.

- $g_j$ was first seen as the result of a $\mathsf{Op}_{sid}$ query. We distinguish two cases.
  - During that $\mathsf{Op}$ query, the observation $ob'$ of result $g_j$ was added to the observation list $\mathsf{Ob}_{sid}$. In this case, $ob'$ must have been processed earlier than $ob$, setting $Rep[g_j]$ correctly in Line 8 (argued inductively).
  - During that $\mathsf{Op}$ query, no observation was added to $\mathsf{Ob}_{sid}$. 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 $\tau_i^{-1}(g_j) = 0 \mod l$, 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 FindRep, then it was set correctly (Claim 3). If $Rep[h^*]$ was not set during the execution of 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,

$$\left| \Pr[\text{EXEC}_{F-wNIZK}, Z, S_{G16}, G-\cdot oGG}(\lambda, z) = 1] - \Pr[\text{EXEC}_{F-wNIZK}, Z, S_{G16}, G-\cdot oSG}(\lambda, z) = 1] \right|$$

$$\leq \frac{9q_1^2 + 3q_1}{p}$$

where $q_1 = m + 3d + 6 + q_Z + 3qp + (\ell + 6)q_V$.

Proof: We count the number of queries made to $G-\cdot oGG/G-\cdot oSG$:

- Initial generation of CRS triggers at most $m + 3d + 4$ queries to OP and 2 queries to TOUCH.
- $Z$ queries $G-\cdot oGG/G-\cdot oSG$ at most $q_Z$ times.
- Each invocation of PROVE triggers at most 3 queries to OP via $S_{G16}$.SIMULATE
- Each invocation of VERIFY triggers at most $\ell + 2$ queries to OP and 4 queries to PAIR via $S_{G16}$.EXTRACT.

In total, at most $q_1 = m + 3d + 6 + q_Z + 3qp + (\ell + 6)q_V$ 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,

$$\left| \Pr[\text{EXEC}_{F-wNIZK}, Z, S_{G16}, G-\cdot oGG}(\lambda, z) = 1] - \Pr[\text{EXEC}_{F-wNIZK'}, Z, S_{G16}, G-\cdot oSG}(\lambda, z) = 1] \right|$$

$$\leq (2q_2 + q_2') \cdot \frac{3q_2'}{2p} + \frac{(9q_2^2 + 3q_2)d_2}{p - 1} + \frac{6q_Z}{p}$$

where $q_2 = q_1$ (see Claim 4), $q_2' = m + 3d + 4 + 3qp$ and $d_2 = 2d - 1$.

Proof: First, let us consider the case where the VERIFY interface receives as input $(x, \pi)$ such that $x$ was previously queried to the PROVE interface and PROVE responded with $\pi$. In $H_1$, $F-wNIZK$.VERIFY$_{sid}(x, \pi)$ always returns 1 since $(x, \pi)$ is guaranteed to exist in the table $T$.

In $H_2$, $F-wNIZK'$.VERIFY$_{sid}(x, \pi)$ also returns 1 since $(x, \pi)$ was simulated by $S_{G16}$ in such a way that it passes the verification condition. Hence, the view of $Z$ is identical in $H_1$ and $H_2$.

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

We now look at the case where the VERIFY interface receives as input $(x, \pi)$ such that $x$ was never queried to the PROVE interface. In $H_1$, $F-wNIZK$.VERIFY$_{sid}(x, \pi)$ outputs 1 if only if $S_{G16}$.EXTRACT$_{sid}(x, \pi)$ successfully outputs $w$ such that $(x, w) \in \mathcal{R}_{QAP}$. In $H_2$,
\(F-wNIZK\). \(\text{Verify}_{\text{sid}}(x, \pi)\) outputs 1 if and only if the verification equation is satisfied by definition. Hence, the view of \(Z\) is identical in \(H_1\) and \(H_2\), except if \(S_{G16}.\text{Extract}_{\text{sid}}(x, \pi)\) in \(H_1\) \text{fails to extract valid witness while} \((x, \pi)\) \text{passes verification. We now bound the probability that this exceptional event occurs in} \(H_1\). \text{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 \(H'_1\), \(G\text{-oSG}\) is extended with additional interfaces \(\text{GETRND}\) and \(\text{COMPUTECONCRETE}\) which can be accessed by a modified simulator \(S'_{G16}\) described in Simulator 2. Note that \(S'_{G16}\) now aborts if one of the session-specific generators is already reserved for another session. That is, \(\text{Touch}_{\text{sid}}(i, g_{\text{sid}, i})\) fails to initialize a fresh variable associated with \(\text{sid}\) if there already exists some \(f\) such that \(\tau_i(f) = g_{\text{sid}, i}\). Since \(Z\) may define \(\tau_i\) for at most 3 new group elements through each query to \(G\text{-oSG}\), the probability that \(S'_{G16}\) aborts is at most \(6q_Z/p\) by the union bound. The view of \(Z\) is identical in \(H_1\) and \(H'_1\) unless \(S'_{G16}\) aborts.

- \(H''_1\): This is identical to \(H'_1\) except that every invocation of \(\text{COMPUTECONCRETE}\) is replaced with \(\text{COMPUTE SYMBOLIC}\) as in the modified simulator described in \(S''_{G16} \)Simulator 3. This transition is justified by Lemma 4. Concretely, counting the number of supplied polynomials to \(\text{COMPUTECONCRETE/COMPUTE SYMBOLIC}:\)
  - Initial generation of CRS supplies at most \(m + 3d + 4\) polynomials.
  - Each invocation of \(\text{PROVE}\) sends at most 3 polynomials via \(\text{SIMULATE}\)
  - Each invocation of \(\text{VERIFY}\) does not trigger any query to \(\text{ComputeX}\)

In total, at most \(q'_2 = m + 3d + 4 + 3q_P\) 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 \(G\text{-oSG}\) 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 \(G\text{-oSG}\) is extended with an additional \(\text{GETREP}_{\text{sid}}\) interface and every invocation of \(\text{FINDREP}\) is replaced with \(\text{GETREP}_{\text{sid}}\) as in the modified simulator \(S'''_{G16}\) 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 \(S'''_{G16}\) is now treated as a formal symbol, the proof
\[ \pi = (A, B, C) \text{ output by } Z \text{ can be expressed as:} \]
\[
\tau^{-1}(A) = p_A(\text{Var}_{-sid}) + \sum_{i=0}^{\ell} A_i (x_\alpha x_\beta x_\gamma) \]
\[
\cdot \left( A_n x_n + A_\beta x_\beta + A_\delta x_\delta + A_\nu x_\nu t(x_\tau)x_\tau^{-1} \right) + \sum_{i=0}^{m} A_i q_i (x_\alpha x_\beta x_\delta) x_\delta^{-1} \]
\[
+ \sum_{j=1}^{Q} A_{\mu(j)} x_{\mu(j)} + \sum_{j=1}^{Q} A_{\nu(j)} (x_{\mu(j)} x_{\nu(j)} - x_\alpha x_\beta - \sum_{i=0}^{\ell} a_{i}^{(j)} q_i (x_\alpha x_\beta x_\delta)) x_\delta^{-1} \]
\[
\tau^{-1}(B) = p_B(\text{Var}_{-sid}) + \sum_{i=0}^{\ell} B_i (x_\alpha x_\beta x_\delta) \]
\[
\cdot \left( B_\delta x_\delta + B_\mu x_\mu + B_\nu x_\nu + B_\tau x_\tau t(x_\tau)x_\tau^{-1} \right) + \sum_{i=0}^{Q} B_i q_i (x_\alpha x_\beta x_\delta) x_\delta^{-1} \]
\[
\tau^{-1}(C) = p_C(\text{Var}_{-sid}) + \sum_{i=0}^{\ell} C_i (x_\alpha x_\beta x_\delta) \]
\[
\cdot \left( C_\delta x_\delta + C_\mu x_\mu + C_\nu x_\nu + C_\tau x_\tau t(x_\tau)x_\tau^{-1} \right) + \sum_{i=0}^{Q} C_i q_i (x_\alpha x_\beta x_\delta) x_\delta^{-1} \]
\]
where \(A_\beta, C_\beta\) are univariate polynomials of degree \(d-2\), \(A_\nu, B_\nu, C_\nu\) are univariate polynomials of degree \(d-1\), \(\text{Var}_{-sid}\) 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 \(\text{GetRep}_{-sid}\). Our goal is show that \((x, w) = (a_i)_{i=1}^{\ell}; \{C_i\}_{i=\ell+1}^{m} \in R_{QAP}\) whenever \(x\) and \(\pi = (A, B, C)\) pass verification.

First, the fact that \(A, B, C\) satisfy the verification condition implies:
\[
\tau^{-1}(A) \cdot \tau^{-1}(B) = \tau^{-1}(C) \cdot (x_{\text{sid,2}} x_\delta) + \sum_{i=0}^{\ell} \left( a_i q_i (x_\alpha x_\beta x_\delta) \right) \]
Focusing on the terms containing the monomial \(x_{\text{sid,1}} x_{\text{sid,2}}\), we have that
\[
(A_n x_n + A_\beta x_\beta + \ldots) (B_\delta x_\delta + B_\mu x_\mu + \ldots) \]
\[
\equiv x_\delta (C_\delta x_\delta + C_\mu x_\mu + \ldots) x_\delta + \sum_{i=0}^{\ell} a_i q_i (x_\alpha x_\beta x_\delta) \]
(1)

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. \(A_{\mu(j)} = B_{\mu(j)} = C_{\mu(j)} = C_{C(j)} = 0\) 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 \(A_{\mu(k)}, B_{\mu(k)}, C_{\mu(k)}\) and \(A_{\mu(j)} = B_{\mu(j)} = C_{\mu(j)} = C_{C(j)} = 0\) for \(j \neq k\), implying that only the \(k\)th 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 \(k\)th

\[\footnote{Although [BKSV21] does not explicitly mention \(A_{\mu(j)}\) and \(C_{\mu(j)}\) (denoted by \(A_{\alpha, j}\) and \(C_{\alpha, j}\) 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 \(B_{\mu(i)} \neq 0\). Then looking at the term involving \(X_{\mu(i)} X_{\delta}\) for \(i \neq j\), we have that \(A_{\mu(i)} B_{\mu(j)} = 0\), implying \(A_{\mu(i)} = 0\) for \(i \neq k\). Looking at the term involving \(X_{\mu(i)} X_{\delta}\), we also have that \(A_{\mu(i)} B_{\delta} - C_{\mu(i)} = 0\) for all \(i \in [Q]\). Thus, \(C_{\mu(i)} = 0\) for \(i \neq k\).} \]
statement \( x^{(k)} = \{q_i^{(k)}\}_{i=1}^\ell \), meaning that the forged proof \( \pi \) is merely a mauled version of \( k \)th simulated proof \( \pi^{(k)} \). In this case, the simulator does not need to extract valid witness as we described before.

\[ \text{Claim 6. Hybrids } H_2 \text{ and } H_3 \text{ are indistinguishable. Concretely,} \]

\[ |\Pr[\text{EXEC}_{F \cdot \text{whizz}}, Z, S_{G16}, G \cdot \text{oSG}(\lambda, z) = 1] - \Pr[\text{EXEC}_{F \cdot \text{whizz}'}, Z, S_{G16}, G \cdot \text{oSG}(\lambda, z) = 1]| \]

\[ \leq 6 \cdot (2q_3 + q_3') \cdot q_3' + \frac{12q_z}{p} \]

where \( q_3 = q_1 \) (see Claim 4) and \( q_3' = m + 3d + 4 + (m - \ell + 4d + 1)q_P \).

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

- \( H_2' \): 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 \( S_{G16} \) aborts if touching the session-specific group generators fails (by checking whether their representation is already defined or not). Moreover, \textbf{Simulate} invokes \textbf{ComputeConcrete} to obtain \( A, B, C \).

- \( H_3' \): 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 \( S_{G16} \) aborts if touching the session-specific group generators fails. Moreover, \textbf{Prove} invokes \textbf{ComputeConcrete} to obtain honestly computed \( A, B, C \) after initializing \( X_r \) and \( X_a \) via \textbf{GetRnd}.

The loss incurred when transitioning to \( H_2' \) from \( H_2 \) is at most \( 6q_z/p \). The same loss applies when transitioning to \( H_3' \) from \( H_3 \). Regarding \textbf{Simulate} of \( H_2' \) as \( B_0 \) and \textbf{Prove} of \( H_3' \) as \( B_1 \), respectively, they indeed satisfy the perfectly equivalence condition as required by Lemma 7 because the joint distribution of \((\tau_1^{-1}(A), \tau_2^{-1}(B), \tau_4^{-1}(C))\) output by both algorithms is identical. To derive the concrete loss, let us count the number of supplied polynomials to \textbf{ComputeConcrete}: 

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

In total, at most \( q_3 = m + 3d + 4 + (m - \ell + 4d + 1)q_P \) polynomials are supplied during an execution of each hybrid. The total number of queries \( q_3 \) to \( G \cdot \text{oSG} \) 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.

\[ \text{Claim 7. Hybrids } H_2 \text{ and } H_4 \text{ are indistinguishable. Concretely,} \]

\[ |\Pr[\text{EXEC}_{F \cdot \text{whizz}'}, Z, S_{G16}, G \cdot \text{oSG}(\lambda, z) = 1] - \Pr[\text{EXEC}_{F \cdot \text{whizz}''}, Z, S_{G16}, G \cdot \text{oSG}(\lambda, z) = 1]| \]

\[ \leq 9q_4^2 + 3q_4 \]

where \( q_4 = m + 3d + 6 + 3q_z + (m - \ell + 4d + 1)q_P + (\ell + 6)q_V \).

\[ \text{Proof: We count the number of queries made to } G \cdot \text{oSG}/G \cdot \text{oSG}: \]

57
Functionality 20: $\mathcal{F}$-wNIZK (used in hybrid $H_2$)

$\mathcal{F}$-wNIZK is parameterized by $\mathcal{R}_{\text{wNIZK}}$, and runs with parties $\mathcal{P}_1, \ldots, \mathcal{P}_N$ and an ideal process adversary $\mathcal{S}_{\text{wNIZK}}$. Moreover, it has direct access to $\mathcal{G}$-oSG. The group operations happening inside Verify are carried out via the corresponding wrapper interfaces of $\mathcal{S}_{\text{wNIZK}}$. It stores proof table $T$ which is initially empty.

\[
\text{INIT}_{\text{wNIZK}}() \quad 1: \quad T \leftarrow \emptyset \quad \text{// Empty table}
\]

\[
\text{PROVE}_{\text{wNIZK}}(x, w) \quad 1: \quad \text{if } (x, w) \notin \mathcal{R} \text{ then return } \bot
\]

\[
2: \quad \pi \leftarrow \mathcal{S}_{\text{wNIZK}}.\text{SIMULATE}_{\text{wNIZK}}(x)
3: \quad T \leftarrow T \cup (x, \pi)
4: \quad \text{return } \pi
\]

\[
\text{VERIFY}_{\text{wNIZK}}(x = \{a_i\}_{i=1}^m, \pi = (A, B, C)) \quad 1: \quad \sigma \leftarrow \mathcal{S}_{\text{wNIZK}}.\text{GETCRS}_{\text{wNIZK}}()
2: \quad C_{\text{pub}} \leftarrow \sum_{i=0}^t a_i q_i (\alpha, \beta, x) \gamma^{-1}
3: \quad \text{return } A \cdot B = C_{\text{pub}} \cdot [\gamma]_{\tau, 2} + C \cdot [\delta]_{\tau, 2} + [\alpha]_{\tau, 1} \cdot [\beta]_{\tau, 2}
\]

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

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

\[
\text{INIT}_{\text{wNIZK}}() \quad 1: \quad T \leftarrow \emptyset \quad \text{// Empty table}
\]

\[
\text{PROVE}_{\text{wNIZK}}(x = \{a_i\}_{i=1}^m, w = \{w_i\}_{i=1}^m) \quad 1: \quad \text{if } (x, w) \notin \mathcal{R}_{\text{wNIZK}} \text{ then return } \bot
\]

\[
2: \quad \sigma \leftarrow \mathcal{S}_{\text{wNIZK}}.\text{GETCRS}_{\text{wNIZK}}()
3: \quad r, s \leftarrow \mathcal{Z}_p
4: \quad \text{Compute } h \in \mathbb{R}^{d-2}(X) \text{ such that } h t = (\sum_{i=0}^m a_i w_i)(\sum_{i=0}^m a_i v_i) - (\sum_{i=0}^m a_i w_i)
5: \quad A := [a]_{\tau, 1} \leftarrow \sum_{i=0}^m a_i u_i(x) + \alpha + \tau\beta_{\tau, 1}
6: \quad B := [b]_{\tau, 2} \leftarrow \sum_{i=0}^m a_i v_i(x) + \beta + \tau\delta_{\tau, 2}
7: \quad C := [c]_{\tau, 1} \leftarrow \sum_{i=0}^m a_i q_i(\alpha, \beta, x) \delta^{-1} + h(x) t(x) \delta^{-1} + sa + rb - r s\delta_{\tau, 1}
8: \quad \text{return } (A, B, C)
\]

\[
\text{VERIFY}_{\text{wNIZK}}(x = \{a_i\}_{i=1}^m, \pi = (A, B, C)) \quad 1: \quad \sigma \leftarrow \mathcal{S}_{\text{wNIZK}}.\text{GETCRS}_{\text{wNIZK}}()
2: \quad C_{\text{pub}} \leftarrow \sum_{i=0}^t a_i q_i (\alpha, \beta, x) \gamma^{-1}
3: \quad \text{return } A \cdot B = C_{\text{pub}} \cdot [\gamma]_{\tau, 2} + C \cdot [\delta]_{\tau, 2} + [\alpha]_{\tau, 1} \cdot [\beta]_{\tau, 2}
\]

- Initial generation of CRS triggers at most $m + 3d + 4$ queries to OP and 2 queries to Touch.
- $Z$ queries $\mathcal{G}$-oSG/$\mathcal{G}$-oSG at most $q_Z$ 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 $\mathcal{F}$-wNIZK''.

In total, at most $q_4 = m + 3d + 6 + 3q_Z + (m - \ell + 4d + 1)qp + (\ell + 6)q_V$ are made during an execution of each hybrid. Plugging $q_4$ into Lemma 3, we obtain the claimed loss. $\blacksquare$
The simulator stores state:
- \( \text{Ob}_\text{ad} \) initially empty lists
- \( \sigma \) Labels for simulated common reference string
- \( \text{td} \) Trapdoor for \( \sigma \)

**GetCRS**, **OP**, **PAIR**, **TOUCH**, **CanonicalGen**, **Observe** are to be called by the environment, while **Simulate** and **Extract** are to be called by \( \mathcal{F} \)-**R1zk**. Note that the interfaces **OP**, **PAIR**, **TOUCH**, **CanonicalGen**, **Observe** are wrappers of the corresponding methods of \( \overline{\mathcal{G}} \)-**Gg2**. We define these so that \( S_{\text{g16}} \) can keep track of all the group operations happening inside the current session.

**Simulator 1: \( S_{\text{g16}} \)**

The simulator stores state:
- \( \text{Ob}_\text{ad} \), \( \text{Ob}^* \) initially empty lists
- \( \sigma \) Labels for simulated common reference string
- \( \text{td} \) Trapdoor for \( \sigma \)

GetCRS\text{ad}() \ commenting only upon creation
1. Run the code for \( \mathcal{F} \)-**CRS**.\text{Init}(). Store \( \sigma \) as \( \text{CRS} \) and \( \text{td} = (x, \alpha, \beta, \delta) \) as a simulation trapdoor, respectively.

GetCRS\text{ad}() 2. return \( \sigma \)

OP\text{ad}(i, g_1, g_2, a_1, a_2)
3. \( k \leftarrow \overline{\mathcal{G}} \)-**Gg2**.\text{OP}(i, g_1, g_2, a_1, a_2)
4. assert \( x \neq \perp \)
5. \text{UpdateOb}\text{ad}((\text{OP}, i, g_1, g_2, a_1, a_2, h))
6. return \( h \)

PAIR\text{ad}(g_1, g_2)
7. \( h \leftarrow \overline{\mathcal{G}} \)-**Gg2**.\text{PAIR}(g_1, g_2)
8. assert \( h \neq \perp \)
9. \text{UpdateOb}\text{ad}((\text{PAIR}, 1, g_1, g_2, h))
10. return \( h \)

TOUCH\text{ad}(i, g)
11. return \( \overline{\mathcal{G}} \)-**Gg2**.\text{TOUCH}(i, g)

CanonicalGen\text{ad}(i)
12. return \( \overline{\mathcal{G}} \)-**Gg2**.\text{CanonicalGen}\text{ad}(i)

Observe\text{ad}()
13. return \( \overline{\mathcal{G}} \)-**Gg2**.\text{Observe}\text{ad}()

UpdateOb\text{ad}(\text{tuple})
14. \( \text{Ob}^* \leftarrow \overline{\mathcal{G}} \)-**Gg2**.\text{Observe}\text{ad}()
15. \( \text{Ob}_\text{ad} \leftarrow \text{Ob}_\text{ad} : (\text{Ob}^* \setminus \text{Ob}^*) : \text{tuple} \)
16. \( \text{Ob}^* \leftarrow \text{Ob}^* \) Stash the current state of observation list stored in \( \overline{\mathcal{G}} \)-**Gg2**
Invoked only upon creation

Identical to those of \( S_{16} \) except they call \( \mathcal{G}^\bullet \mathcal{O}_G \). We overload the bracket notations within \( S_{16} \) as follows, assuming that symbolic variables \( X_i, X_{i+1}, \ldots \) are obtained through \( \mathcal{G}^\bullet \mathcal{O}_G \):

\[
[f(x, y, \ldots)]_{\text{out}, i} := \mathcal{G}^\bullet \mathcal{O}_G(\text{Extract}(i, g_{\text{out},i}, f(X_i, X_{i+1}, \ldots)))
\]

\( \text{Init}_{\text{out}}() \) Invoked only upon creation

\[
\begin{align*}
\text{for } i &= 1, 2 \text{ do} \\
2: & \quad g_{\text{out},i} \in S_i \\
3: & \quad \mathcal{G}^\bullet \mathcal{O}_G(\text{Touch}_{\text{out}}(i, g_{\text{out},i})) \\
4: & \quad \text{if } g_{\text{out},i} \in \text{im}(r_i) \text{ then abort} \\
5: & \quad td := \{X_i, X_{i+1}, X_{i+2}, X_{i+3}\} \leftarrow \mathcal{G}^\bullet \mathcal{O}_G(\text{GetRnd}_{\text{out}}()) \\
6: & \quad \sigma_1 \leftarrow [\alpha, \beta, \gamma, \delta, x_{1,2}, x_{1,3}] \\
7: & \quad \sigma_2 \leftarrow [\alpha, \beta, \gamma, \delta, x_{1,2}, x_{1,3}] \\
8: & \quad \sigma \leftarrow (\sigma_1, \sigma_2)
\end{align*}
\]

\( \text{Simulate}_{\text{out}}(x = \{a_i\}_{i=1}^t) \)

\[
\begin{align*}
X_i, X_{i+1} & \leftarrow \mathcal{G}^\bullet \mathcal{O}_G(\text{GetRnd}_{\text{out}}()) \\
A & \leftarrow [\gamma]_{\text{out},2} \\
B & \leftarrow [\eta]_{\text{out},2} \\
C & \leftarrow [\alpha \eta - \alpha \beta - \sum_i a_i q_i(\alpha, \beta, x) \delta^{-1}]_{\text{out},1} \\
\text{return } (A, B, C)
\end{align*}
\]

Identical to \( S_{16} \) except the \( \text{Extract} \) interface

\[
\text{Extract}_{\text{out}}(x = \{a_i\}_{i=1}^t, \pi = (A, B, C))
\]

\[
\begin{align*}
1: & \quad C_{\text{pub}} \leftarrow \sum_i a_i q_i(\alpha, \beta, x) \gamma^{-1} \mid_{\text{out},1} \\
2: & \quad \text{if } A \cdot B \neq C_{\text{pub}} \cdot [\gamma]_{\text{out},2} + C \cdot [\beta]_{\text{out},2} + [\alpha]_{\text{out},1} \cdot [\beta]_{\text{out},2} \text{ then junk} \\
3: & \quad \text{return junk} \\
4: & \quad g \leftarrow \mathcal{G}^\bullet \mathcal{O}_G(\text{CanonicalGen}_{\text{out}}(1)) \\
5: & \quad B \leftarrow (g, [\eta]_{\text{out},1}, [q_i(\alpha, \beta, x) \delta^{-1}]_{\text{out},2} \mid_{\text{out},1}) \\
6: & \quad \{\cdot, \cdot\}_{\text{out},2} \leftarrow \mathcal{G}^\bullet \mathcal{O}_G(\text{GetRnd}_{\text{out}}(1, C, B)) \\
7: & \quad w \leftarrow [C_{\text{pub}}]_{\text{out},2} \\
8: & \quad \text{if } (x, w) \in \mathcal{N}_{\text{out}} \text{ then} \\
9: & \quad \text{return } w \\
10: & \quad \text{else} \\
11: & \quad \text{return maul}
\end{align*}
\]