Aalto University
School of Science

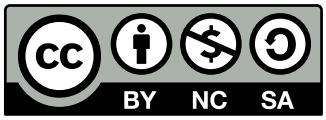
Master's Programme in Security and Cloud Computing (SECCLO)

Towards formally verifying the TLS 1.3 Key Schedule Security in SSBee

Amirhosein Rajabi

Master's Thesis
2025

Research reported in this publication was supported by the Research Council of Finland grant 35895 and the Amazon Research Award *Secure Messaging: Updates, Efficiency & Verification*, Fall 2023.



Aalto University
School of Science

Author Amirhosein Rajabi

Title Towards formally verifying the TLS 1.3 Key Schedule Security in SSBee

Degree programme Security and Cloud Computing (SECCLO)

Major Security and Cloud Computing

Supervisor Prof. Chris Brzuska, Prof. Sebastian Alexander Mödersheim

Advisor Prof. Christoph Egger

Date 15 July 2025 **Number of pages** 194 **Language** English

Abstract

TLS is one of the most important cryptographic protocols protecting communications on the Internet. The standardization process of TLS 1.3 prioritized addressing security issues over implementation concerns. Early drafts of TLS 1.3 standard welcomed inspiration from the cryptography community and specifically the OPTLS handshake by Krawczyk and Wee. TLS 1.3 key schedule is at the core of the TLS 1.3 handshake responsible for all cryptographic operations deriving keys for the handshake and record layer, such as handshake and application traffic secrets, among others.

The main security requirement for the key schedule is to generate pseudorandom and unique keys given honest Diffie-Hellman shares or Pre-shared Key (PSK), or both (depending on the key exchange mode). Brzuska, Delignat-Lavaud, Egger, Fournet, Kohbrok, Kohlweiss (BDEFKK) (Asiacrypt, 2022) model these security goals and reduce the security of the TLS 1.3 standard key schedule (as appeared in the standard) to the security assumptions of the underlying primitives. BDEFKK capture the security properties of the key schedule in the State-separating Proofs (SSP) framework. SSP is one of the compositional frameworks in cryptography developed by the community for modular security analysis. SSP enables BDEFKK to break the complexity of the key schedule security games into several small components each with only one responsibility. However, their modular analysis is still long and over 100 pages which makes human verification difficult.

This thesis brings to light that formal verification of large scale proofs of real world protocols' cryptographic analysis is indeed approachable, provided the analysis itself utilizes modular analysis and verification-friendly proof framework that makes the gap with verification tools smaller. Specifically, we lay the foundations, formalize the security games, and take steps towards verification of one subtle lemma from the analysis of BDEFKK, pen-and-paper proof of which takes 10 pages of the paper, in SSBee. SSBee is a novel software toolchain (and at the time of writing this thesis, being actively developed by Chris Brzuska, Christoph Egger, and Jan Winkelmann) tailored particularly for formalizing reduction proofs written in the SSP framework. We successfully verify major statements in the lemma and find and add missing or minor details in the proof. Additionally, we highlight verification techniques discovered during different stages of the project. We conclude our verification story with our vision for future of SSBee with promising integrations with other tools for program verification.

Lastly, BDEFKK reduce the security of the key schedule to a new Salted Oracle Diffie-Hellman (SODH) assumption among others. Inspired by mmPRF-ODH assumption instantiation of Brendel, Fischlin, Günther, and Janson, we analyze the assumption in the random oracle model and partially reduce to the well-known Strong Diffie-Hellman (SDH) assumption introduced by Abdalla, Bellare, and Rogaway and formally verify one non trivial step of the reduction with SSBee. Although BDEFKK were more optimistically claiming that SODH assumption can be reduced to Computational Diffie-Hellman (CDH) assumption, this thesis explains why it seems too optimistic.

Keywords TLS 1.3 Key Schedule, Formal Verification, SSBee, Invariant Argument, Code Equivalence, Salted Oracle Diffie-Hellman Assumption

Acknowledgements

I want to thank all my supervisors and advisors Chris Brzuska, Christoph Egger, and Sebastian Mödersheim (in alphabetical order) for guiding me during the whole process. Special thanks goes to Chris who retaught me Cryptography and mentored and supported me during my whole master's studies. I am grateful for the time and effort you put in our discussions, all your feedback, the insights you gave me, as well as answering my late hour and weekend long messages. Thank you for having faith in me. I would like to thank Sebastian for all the fruitful discussions we had and the experiences in Formal Verification that you shared with me. Thank you Christoph for giving feedback offline or on Zoom for all my informally written proofs as well as your amazing insights in modeling TLS security games in SSBee. Thank you all for having me as a student.

I want to extend my thanks to Christoph Matheja and the rest of Program Verification teaching team at DTU who introduced me to the Formal Verification world through their amazing course and answering my questions on Teams. I also want to thank my Program Verification course project teammates whose ideas in the project were applicable in my thesis verification journey and indirectly affected it. Moreover, I want to thank Jan Winkelmann for our discussions on SSBee and alternative solutions to current SSBee limitations. I want to thank Kirthivaasan Puniamurthy for all our crypto-related and non-crypto discussions.

I want to thank my friends (SECCLO, those in Iran, DTU, and elsewhere) who had to listen to my thesis even in the middle of Finnish national parks. Thank you for your feedback on my presentations. I even got inspirations and found mistakes in my thesis after your questions in the woods. Thank you for all the fun atmosphere you created. Spending time with you all in person and remotely on meetings kept me sane and alive.

Last but not least, I want to express my sincerest gratitude to my family and loved ones (my mom, dad, sister, uncle Reza and his family, grandparents). Without your support, literally from my very first days, and your encouragement and motivation in science, I wouldn't have been where I am, career-wise, geographically, mentally, and physically.

Helsinki, 15 July 2025

Amirhosein Rajabi

Co-funded by
the European Union

Contents

1 Introduction

Transport Layer Security (TLS) is a widespread protocol on the Internet that establishes a secure authenticated channel between Internet users. After many vulnerabilities, exploits and attack proof-of-concepts were reported for earleir versions of TLS (SSL X.0, TLS 1.x, TLS 1.2), TLS working group at IETF started standardization process of TLS 1.3 in 2014. In February 2015, IETF published an information RFC with a list of known attacks on TLS/SSL, including renegotiation attacks [CVE09], downgrade attacks (FREAK [BBDL+15], Logjam [ABD+15]), cross-protocol attacks (DROWN [ASS+16]), padding oracle vulnerabilities of constructions based on CBC mode (BEAST [AP13], POODLE), web cookie recovery when data compression in place (CRIME) and HTTP compression (BREACH), etc. The new Internet security standardization brought security concerns upfront and benefitted from results in cryptographic research and community as well as attacks found by automated tools. Therefore, security researchers and cryptographers contributed to the standardization by analyzing early drafts as well as latest version of TLS 1.3 on paper ([DFGS20, BDLE+21, Bla18, FG17, DG19]) and with automated tools (such as Tamarin [CHH+17, CHSvdM16] as well as CryptoVerif and ProVerif [BBK17]) and gave feedback through the public mailing list of IETF.

1.1 Symbolic model vs computational model

Security analysis of cryptographic protocols (such as ones appeared in TLS 1.3 standardization drafts) can be roughly divided into two groups of computational and symbolic models. Analysis performed with automated tools such as Tamarin [BCDS22] and ProVerif [Bla16] are in the symbolic model and roughly model cryptography as a black-box. In the symbolic model, attackers can not break cryptography but can read, write, delete, intercept messages in the network (Dolev-Yao mdoel), cryptographic algorithms are considered to be mathematical functions, and bitstring messages to be abstract terms, among other differences. This brings security guarantees up to the level of abstractions used in the model. Symbolic tools help with modeling security properties of the protocols built out of cryptographic primitives and automatically finding complex attacks that are difficult for humans to find. (See 18-message attack on post-handshake client authentication found by Tamarin [CHH+17].)

Although symbolic models can be extended to consider other properties of cryptographic primitives and algebraic operations (e.g. finite field operations), extensive and more complex models come with the risk of nontermination of the tools due to undecidable nature of the problem and a huge search space. It is worth mentioning that recent improvements in these tools and new symbolic models for signatures [JCCGS19], hash functions [CCD+23], Authenticated Encryption with Associated Data (AEAD) [CDJZ23], and prime and nonprime-ordered groups [CJ19] have led to discovery of more attacks in more extensive models even beyond Dolev-Yao model.

On the other hand, symbolic tools are not suitable for *reduction-based* analysis that is usually performed by cryptographers on paper. Reduction-based analysis used by cryptographers considers computational model which restricts the compu-

tational resources of the adversary. Namely, cryptographers consider adversaries to be probabilistic polynomial-time algorithms who can not solve hard computational/-mathematical problems and build protocols and schemes that any successful exploit of their security goals imply an efficient algorithm (achieved through security reductions) for solving these hard problems. (cf. idealized adversaries in symbolic model who can not break any cryptography) Security guarantees provided by the computational model are stronger than the symbolic model but is much more difficult to produce and verify by humans. We refer the reader to [Bla12] for a more detailed comparison of symbolic and computational model. At the end of the day, both types of analysis are necessary for real-world protocols to achieve protection against known attacks and build more confidence for the lifetime of the protocol. [1]

1.2 State-separating proofs (SSP) framework

The cryptography community has developed many proof frameworks in computational model for analyzing of complex cryptographic protocols, avoiding mistakes in security reductions, and reducing the gap with formal verification tools. State-separating proofs (SSP) [BDLF+18, Koh23] is one of compositional frameworks in cryptography among Universal Composability [Can00] introduced by Ran Canetti for composition of key exchange protocols and symmetric key encryption, Abstract Cryptography [MR11] by Maurer and Renner for a top-down and axiomatic approach to security definitions, Constructive Cryptography [Mau11] by Maurer as an application of Abstract Cryptography to cryptographic primitives as well as educational purposes, etc. In addition to TLS 1.3 key schedule and key exchange security, SSP has been used for cryptographic analysis of Message Layer Security (MLS) [BCK21] and Yao's garbling scheme [BO21]. SSP, inspired by the previous frameworks and process algebra, allows modular security analysis. SSP requires proper decomposition of monolithic code-based Bellare-Rogaway-style security games into a collection of modular stateful components called packages. Analogous to classes in object-oriented programming, SSP packages contain a set of oracles with their code-based definitions. All oracles of the packages have access to and share the private state of the package such as tables and variables. The package oracles, though, are exposed to other packages as the package output interface. Packages also require a set of oracles for their operation as their input interface, which their oracles call and depend on. Security games are defined as composition of packages and naturally induce a call graph as a directed acyclic graph. In contrast to experiment-based security definitions, adversaries interact with oracles exposed by SSP games and their success probability in different games is analyzed. Security proofs then consist of two types of steps: reduction and game

[1] As described by [BBB+19], symbolic and computational models provide confidence in different levels of abstraction and different scopes of the real world implementation. Symbolic model scope is larger with higher level of abstraction (hence lower confidence) while the computational model scope is smaller with lower level of abstraction (hence higher confidence). Combined together, we get higher assurance of the cryptographic system in the threat model. It also worths mentioning that program verification of actual code implementation of cryptographic protocols is complementory to previous approaches.

(code) equivalence. (We use the terms "code equivalence", "game equivalence", and "functional equivalence" interchangeably.) Visualizing cryptographic reductions, reduction steps are essentially cuts in the the game call graph by cutting out the security game of a cryptographic primitive (to which we reduce security) from the call graph of enclosing security game and leaving the rest packages as the reduction package. Game equivalence steps show that input-output behavior of two games are the same or, in other words, they are functionally equivalent. (i.e. they are indistinguishable even by an unbounded adversary) Any SSP proof has at least two code equivalence steps to show that the real (ideal) monolithic version of the game (before decomposition into packages) is equivalent to the real (ideal) modular version of the game (after decomposition into packages). In Section 2.1, we formally define SSP and give a security reduction in SSP as an example. The example reduces IND-CCA security of a hybrid public key encryption scheme (KEM-DEM paradigm) by Cramer and Shoup [CS98].

1.3 Formal verification in the computational model

Several formal verification tools exist for cryptographic analysis in computational model, such as CryptoVerif, EasyCrypt, ProofFrog, and SSBee. CryptoVerif [BJ23] is one of these tools and its input language is very similar to ProVerif, which allows to directly describe protocols and prove properties about them by reducing to computational assumptions. EasyCrypt [BGHB11] is an interactive theorem prover focusing on verification of security reductions in code-based game-playing proofs. EasyCrypt uses a language closer to how cryptographers use to define their security games. Based on probabilistic relational Hoare logic [BGZB09], EasyCrypt has strong foundations for reasoning about probabilistic security games and probability distributions they induce. However, as a interactive theorem prover, it still needs significant interaction with the user to guide through each step of the proof. ProofFrog [EMS25, Eva24] is an automated tool for verifying cryptographic hybrid arguments and unlike EasyCrypt, it does not need much interaction with the user to prove security reductions. In fact, the user only needs to define security games, reductions, and security assumptions and ProofFrog can identify required hybrid games and perform automatic game transformations and verify the real game can be reached from the ideal game through several game hops.

SSBee [BEW25] is a novel toolchain and at the time of writing this thesis is being developed by Chris Brzuska, Christoph Egger, and Jan Winkelmann. SSBee is designed to automate proof steps in SSP framework (reductions and game equivalence). SSBee defines a new language for the user to write the code of their packages, security games as composition of packages, security proofs including real and ideal games, assumptions, and intermediary (hybrid) games for game-hopping proofs with an indication of game hop types (i.e. reductions or game equivalence) and relevant hints. SSBee compiler performs a type checking on all input files (particularly oracle codes) and algorithmically checks SSP-style reduction steps that comes for free as a result of formalization effort in SSP. It then verifies game equivalence steps by compiling oracle codes to SMT-LIB language and using a Satisfiability Modulo Theory (SMT)

solver to verify relevant proof obligations. SSBee needs minimal interaction from the user by requiring the user to state some hints for the verification of code equivalence steps. These hints are essentially state relations (between the games to be proved equivalent) and are called "invariants". SSBee tries to prove these hints too while using them for other code equivalence proof obligations. We illustrate invariants together with other key concepts in SSBee in Section 2.5 using the same formalization of KEM-DEM example in SSP explained in Section 2.1. Section 2.5 also presents a tutorial on verification of KEM-DEM security reduction in SSBee.

In addition to SSBee, SSProve [HRM+21] is a formalization effort of SSP in Rocq interactive theorem prover [Roc25]. SSProve has stronger formal foundations as a result of being based on Rocq but requires more interaction with the user while SSBee uses SMT solvers to automate some proof steps with minimal interaction with the user. Dupressoir, Kohbrok, and Oechsner [DKO21] formalized SSP in EasyCrypt with a natural correspondence of SSP packages with EasyCrypt modules. They derive a formal security proof of Cryptobox family of public-key authenticated encryption.

Compared to EasyCrypt, SSBee has limited features, such as limited support for randomness samplings and probability distributions, which make automatization simpler with native support for SSP proofs. When compared to ProofFrog, SSBee has no support for automatic game transformation and any two equivalent games that can be transformed to each other using code transformation (i.e. no computational game hop) shall be proved as a separate code equivalence step or as a reduction to two statistically equivalent games. [2] However, reductions come for free as result of proper formalization in SSP. (SSBee does not directly support statistical game equivalence due to lack of support for probability distributions. Refer to Section 5, though, for an example of an effort to formalize a statistical hop in SSBee.)

1.4 TLS 1.3 Key Schedule Security

Early drafts as well as latest version of TLS 1.3 have been analyze in computational model on paper. DFGS [DFGS20] analyzed handshake of TLS 1.3 in each of three key exchange modes supported in isolation. (We introduce key exchange modes in Section 3.) Egger [Egg22] analyzed key exchange security of TLS 1.3 standard [Res18] in its latest version using state-separating proofs framework (SSP) and reduce the security of key exchange to that of key schedule. Key schedule is at the core of TLS 1.3 standard handshake protocol which derives all keys required for encryption and authentication of handshake as well as record layer messages [3] given the pre-shared key (PSK) and/or Diffie-Hellman (DH) secret key. Key schedule security requires that generated keys are pseudorandom and unique if at least one of the input key material (PSK or DH secret) were honest. (not compromised by the adversary) Brzuska, Delignat-Lavaud, Egger, Fournet, Kohbrok, Kohlweiss (BDEFKK) [BDLE+21] have reduced TLS 1.3 standard key schedule security to security assumptions of primitives used in the protocol (i.e.

[2] It is, though, an interesting line of research to integrate the tools for more automatization of game equivalence.

[3] It also generates exporter keys to be used by the application layer.

collision resistance of SHA series hash functions, (dual) pseudorandomness and pre-image resistance of HMAC, and Salted Oracle Diffie-Hellman (SODH)). This approach (together with concepts from SSP) has made the key exchange proof modular and composable, considering the standard being over 100 pages. Moreover, modeling of BDEFKK captures all key exchange modes of TLS 1.3 at the same time. BDEFKK also prove, in the same model, pseudorandomness and uniqueness of resumption PSKs computed by the key schedule from resumption master secret in the previous session, in contrast to analysis of DFGS who do not model session resumption. We give an overview of the TLS 1.3 key schedule and security analysis in Section 3.

1.5 Original Research Questions

Considering the size and complexity of their proof, thesis started with the goal to determine whether SSBee as a new automated tool can be used for verification of TLS 1.3 key schedule security of [BDLE+21] and if so, present a machine-checked proof of the analysis. We were also interested in possible new features for SSBee and finding bugs from the tool as well as mechanisms to make SSBee more user friendly as a tool in a working cryptographer's toolbox. Moreover, it was unknown whether the analysis as is can be modeled in the tool and already existing techniques in the proof and SSBee suffice for verification or additional ideas are needed for both modeling and verification. Furthermore, in case of a failure in verification, we were interested in finding obstacles to automatization of the proof and how the proof can be made more SMT-prover friendly (or even what improvements to SSBee are needed). Finally, we wanted to complement state relations described in one of the major code equivalence game hops of [BDLE+21] and possibly find all missing or imprecise state relations (if any) described in the paper.

1.6 Contributions

This thesis take steps towards formal verification of TLS 1.3 key schedule security reduction of [BDLE+21] in SSBee. BDEFKK prove two subtle game equivalence steps, which, respectively, introduce a mapping package, followed by security reductions to modular assumptions, and remove the mapping from the output keys. We explain the intuition and details of mapping package in Section 3. Nevertheless, BDEFKK import a new proof technique (state relations proved to be invariants) from program verification to cryptography to prove these two code equivalence steps. (The same technique is used by SSBee to discharge proof obligations for game equivalence steps.) Although the resulting proof is rigorous, performing code analysis on a per-and-paper proof is error-prone, lengthy, and hard-to-verify for humans. [4]

We focus on Lemma C.2 of [BDLE+21] in this thesis and in order to verify the code equivalence in SSBee, we need to define security games and packages in the language of SSBee. Limited features of SSBee language at the time of writing this thesis, such as no support for loops or recursive data types, brought challenges but did

[4] The proof of lemma C.2 takes 10 pages. Proof of Lemma C.5 takes up 5 pages.

not prevent the formalization process. For each of these cases that direct translation of SSP code to SSBee language is not possible, we explain our encoding and argue its soundness. Moreover due to the size of security games in key schedule, we even automatize generation of call graph of security games (essentially composition of packages) with a script. After preparing the security games and packages, we move to writing the security proof and make progress with one subgoal of code equivalence at a time. During the verification process, we speed up the SMT solver and make proof debugging easier by introducing additional lemmata and invariants. SSBee allows us to also prove these additional lemmata. We also discovered and proved "Invariant bubbling" theorem that roughly states properties proved to be correct in subgames are automatically transferred (bubbled up) to enclosing games. Ironically, it allows to prove stronger claims in enclosing games with less effort. [5] This technique can save time both from the proof writer and SMT solver, although is not yet implemented as a formal technique in SSBee but it is our vision to add it in the future. We apply the theorem to an important subgame appearing in the key schedule security game.

Furthermore, we encountered some limitations of cvc5 (the SMT solver currently supported by SSBee) in specific problems, such as universal quantifier instantiation. We mention one case explicitly and our workaround for the issue both in Sections 2.5 and 4. It is, though, still not clear what other limitations they might bring to the table and it is an interesting research question to find more corner cases. (possibly over other code equivalence steps in the paper.)

Finally, we can confirm the claims in analysis of BDEFKK are plausible and upto our current status of verification perfectly correct except for missing minor details. However, we found missing checks causing trivial attacks as a side-product of formal verification in the code of some packages of the security model. We, though, confirmed that these checks are indeed included in the TLS 1.3 standard preventing the attacks.

BDEFKK introduce Salted Oracle Diffie-Hellman (SODH) assumption in their TLS 1.3 key schedule security analysis. SODH is a new variant of Oracle Diffie-Hellman assumption, first appeared in the analysis of DHIES encryption scheme by Abdalla, Bellare, and Rogaway [ABR01]. BDEFKK, however, only mention without proof that SODH can be reduced to computational Diffie-Hellman (CDH) assumption in the random oracle model. In Section 5, we analyze SODH assumption in the random oracle model and partially reduce to strong Diffie-Hellman assumption (SDH), which is the same as CDH except the adversary has additionally access to a DDH-like [6] oracle. Abdalla, Bellare, and Rogaway [ABR01] also presented SDH in their analysis of DHIES and argued plausibility of the assumption in the generic group model.

1.7 Outline

Section 2 formally defines SSP with the running example of the KEM-DEM paradigm and builds up the foundations with key concepts from SSBee and SMT solvers for a

[5] Such properties are stronger because they hold in all enclosing games that contain the packages composed exactly in the same way as in the given subgame.

[6] decisional Diffie-Hellman

tutorial on verification of KEM-DEM CCA security reduction in SSBee. Section 3 discusses the necessary background from TLS 1.3 handshake protocol, supported key exchange modes, key schedule structure, simulation-based security definition of key schedule, and conclude with an overview of key schedule security reduction. Section 4 reports the verification process and highlights proof/debugging techniques/tips for future users of SSBee. Finally, Section 5 presents the analysis of SODH assumption.

2 Preliminaries

This section begins with formal definition of State Separating Proofs (SSP) framework and uses KEM-DEM as a running example. We then present an overview of SMT solvers and SMT-LIB language and introduce key concepts from SSBee using the KEM-DEM example. We demonstrate how the KEM-DEM security reduction can be verified in SSBee. The material in this chapter lay down the cryptography and verification background required for the following chapters of the thesis.

2.1 State Separating Proofs (SSP)

State Separating Proofs (SSP) framework [BDLF+18] is a compositional framework in cryptography and a variant of code-based game-playing proofs framework introduced by Bellare and Rogaway (BR) [BR04]. Traditionally, security definitions in cryptography analyze the winning probabiliy of an adversary interacting with a challenger in a security game, known as game-playing definitions. Usually the adversary is also given access to some oracles that it can freely query. Bellare and Rogaway formalized this setting in their seminal work such that security games can be expressed concretely with their pseudocode, and, hence, the name of framework code-based game-playing definitions or proofs. (They additionally proved the indeed fundamental lemma of game-playing which relates the advantage of an adversary with a bad event happening in the game.)

Essentially, in BR-style games, the challenger experiment is expressed with its pseudocode as initialization and finalization procedures. More importantly, oracles that adversary has access to are also defined as procedures with their pseudocode and clear input and output interfaces. Then the advantage of adversary is analyzed when it is interacting with a real or an ideal game.

In contrast to BR-style games, SSP wishes to defines games as a collection of oracles exposed to the adversary, requiring challenger, initialization, and finalization functionalities to be part of oracles. We then analyze the advantage of adversary interacting with the the oracles of real and ideal games. SSP defines *packages* for the sake of formalization.

Definition 2.1 (Packages)**.** A package M is a collection of oracles (algorithms) with their pseudocodes that can share some private internal state of the package while modifying and reading from the state upon calls to their oracles but can't expose access to their state unless through oracle calls. Packages have a clear ouput interface $\mathsf{out}(M)$, a set of oracle signatures they expose to other packages and a clear input interface $\mathsf{in}(M)$, a set of oracle signatures they depend on for their functionality (possibly exposed by other packages).

Remark. We refrain from precisely defining pseudocodes and package languages. Refer to the work of [BDLF+18] for these definitions and other properties they prove about packages and their compositions when they introduce SSP. For the sake of this thesis and many usecases in cryptography, one can think of the pseudocode langauge

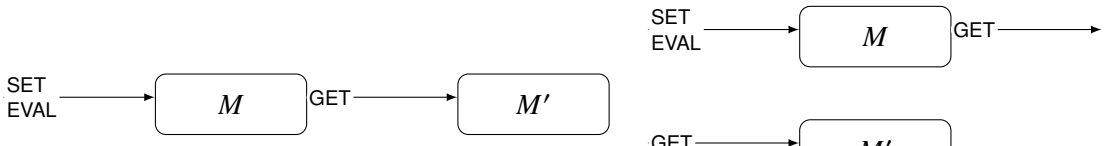
has statements for randomness samplings from distributions and usual control flow statements as well as data storage mechanisms, such as tables, sets, lists, variables, etc.

Packages and classes in object oriented programming have a close relationship. Classes consists of several methods and have a clear output interface for the methods they expose to other classes and a clear dependency on methods they call from other classes. Notice that package states are similar to class private fields that can not be shared with other packages (classes) except through oracle (method) calls. Therefore, they are state separating.

Definition 2.1 brings restrictions and simaltanuously possibility of package compositions. In a quick comparison with BR games, oracles can be thought of procedures and packages can be considered games without initialization and finalization code, which is very rare outisde of SSP world where adversaries interact with a challenger who execute code before and after calling the adversary. However, SSP packages lift any restrictions on type of oracles and allow package consumers to call any oracles in any order. Therefore, oracles should take care of the possibility of being called in any order and respond accordingly. On the other hand, packages can be composed, analogous to classes. This enables modulairty and inspired by SOILD object oriented design principles, one can design packages with single responsiblity.

Definition 2.2 (Package composition)**.** Package $M \circ M'$ (also $M \rightarrow M'$) with output interface $\mathsf{out}(M)$ and input interface $\mathsf{in}(M')$ is the *sequential* composition of packages M and M' by inlining code of callee oracles of M' in the code of caller oracles of M provided that the input interface of M matches output interface of M' (i.e. $\mathsf{in}(M) \subseteq \mathsf{out}(M')$). Package $M|M'$ or $\frac{M}{M'}$ with output interface $\mathsf{out}(M) \cup \mathsf{out}(M')$ and input interface $\mathsf{in}(M) \cup \mathsf{in}(M')$ is the *parallel* composition of packages M and M' by taking union of state variables and oracles of M and M' provided that the output interfaces and state variables of M and M' are disjoint.

Figure 1a shows graphical representation of a sequential composition of packages M and M' where $\mathsf{out}(M) = \{\mathsf{SET}, \mathsf{EVAL}\}$, $\mathsf{in}(M) = \mathsf{out}(M') = \{\mathsf{GET}\}$, and $\mathsf{in}(M') = \varnothing$. Figure 1b shows a parallel composition of packages M and M' with similar interfaces. The main idea is to allow package compositions to induce a call graph similar to dependency graphs of classes.



(a) Sequential composition $M \circ M'$ (also $M \xrightarrow{\mathsf{GET}} M'$)

(b) Parallel composition $\frac{M}{M'}$

Figure 1: Package compositions

Sequential compositions simply mix the state of packages (if they are disjoint) and inline the code of callee oracles in the caller oracles. Figure 2 shows an example of code inlining as a result of sequential composition. Parallel composition also mix

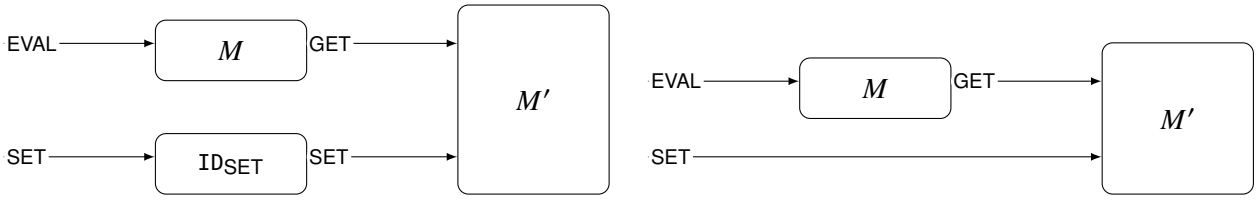
the state of packages (if they are disjoint) but combine the output interfaces of the packages if they do not overlap.

$\underline{\underline{M}}$	$\underline{\underline{M'}}$	$\underline{\underline{M \circ M'}}$
State	State	State
c : challenge	k_1 : left key	k_1 : left key
	k_2 : right key	k_2 : right key
		c : challenge
⋮	⋮	⋮
EVAL(m)	GET()	EVAL(m)
$k \leftarrow$ GET()	**return** $k_1 \| \| k_2$	$k \leftarrow k_1 \| \| k_2$
return $k \oplus m$	⋮	**return** $k \oplus m$
⋮		⋮

Figure 2: Code inlining in sequential composition

Definition 2.3 (Identity package)**.** An identity package $\mathtt{ID}_S$ with a set of oracles S is package with input and output interfaces $\mathsf{in}(M) = \mathsf{out}(M) = S$ that forwards every call to its exposed oracle $O \in S$ to the corresponding oracle provided through its input interface.

We usually describe complex package compositions by the call graph of the packages without specifying the sequential or parallel composition. However, we can formalize the call graph using the identity package. For example, an informal package composition as in Figure 3b is formally a combination of sequential and parallel composition described in 3a.



(a) Formal visualization of package **(b)** Informal visualization

Figure 3: Visualization of package $\frac{M}{\mathtt{ID}_{\mathsf{SET}}} \circ M'$ (also $\frac{M}{\mathtt{ID}_{\mathsf{SET}}} \rightarrow M'$)

Definition 2.4 (Games and adversaries)**.** A game is a package G such that $\mathsf{in}(G) = \varnothing$. An adversary $\mathcal{A}$ playing in (or interacting with) a game G is a package with an input interface the same as output interface of the game (i.e. $\mathsf{in}(\mathcal{A}) = \mathsf{out}(G)$) and an output interface $\mathsf{out}(\mathcal{A}) = \{\mathsf{run}\}$ where $\mathsf{run}()$ returns either 0 or 1.

Since games and composition of adversaries with games ($\mathcal{A} \to G$) are packages with empty input interface, each of their oracles have a well-defined behaviour when called. Therefore, we can analyze the probability of $\mathsf{run}()$ oracle of adversary (composed with G) returning 1 denoted by $\Pr[1 = \mathcal{A} \to G]$.

Definition 2.5 (Adverserial advantage)**.** Let G^0 (real) and G^1 (ideal) be two games where $\mathsf{out}(G_0) = \mathsf{out}(G_1)$. If $\mathcal{A}$ is an adversary such that $\mathsf{in}(\mathcal{A}) = \mathsf{out}(G_0) = \mathsf{out}(G_1)$, we read the adversary is against games G^0 and G^1. We also define advantage of an adversary against games G^0 (real) and G^1 (ideal) as follows:

$$\mathsf{Adv}(\mathcal{A}, G^0, G^1) = \mathsf{Adv}(\mathcal{A}, G^b) := |\Pr[1 = \mathcal{A} \to G^0] - \Pr[1 = \mathcal{A} \to G^1]|$$

Informally, we might read the advantage of adversary distinguishing the real game G^0 and ideal game G^1 (or game pair G^b) is $\mathsf{Adv}(\mathcal{A}, G^b)$.

In the rest of the thesis, we often define pair of real and ideal games at the same time using an idealization bit b as a parameter for the game and packages in the pseudocode of oracles. Nevertheless, games can have other parameters, such as a security parameter λ. Additionally, we are mostly dealing with concrete security setting, in contrast to asymptotic setting [Rog06, BL12]. Namely, we concrelty relate (bound) the advantages of adversaries in our security games to advatnages of new adversaries in some security assumptions. We don't explicitly refer to the security parameter λ and prove the advantage is a negligible function of λ or adversaries being polynomial time with respect to λ. Instead we bound the advtange of adversaries for example with the concrete number of queries to specific oracles. Running time of reductions that construct new adversaries can be easily related to the original adversaries using their pseudocode. Games will also have concrete parameters, such as a specific hash function or group description, for which we have standard security assumptions.

If the advantage of even unbounded adversaries (i.e. running in more than polynomial time) when distinguishing two games G and H is zero, we call the games to be code equivalent or functionally equivalent or perfectly indistinguishable and denote it by $G \overset{code}{\equiv} H$.

Notice that literature sometimes refer to each of the terms *code equivalent*, *functionally equivalent*, or *perfectly indistinguishable* in different scenarios, although the end goal is the advantage of adversary being zero. [BDLF+18] chooses the term "perfect indistinguishability" when the advantage is zero but "code equivalence" when code of the oracles of the games can be transformed to each other using code inlining or variable renaming. Therefore, code equivalence implies perfect indistinguishability. Moreover, functional equivalence refers to the scenario when games do not have exactly the same code (upto renaming) but have the same input-output behaviour. We will interchangably use either of these terms for the same concept but benefit from the "same input-output behaviour" interpretation in the rest of thesis and encourage the reader to think about the "zero advantage under unbounded adversary" as such. We will elaborate on this topic in Section 2.5.

Finally, Lemma 2.1 gives the sufficient conditions for a general reduction of a complicated and complex security game pair G^b_{big} (usually a scheme or construction) to a simpler security game pair G^b_{small} (usually a primitive). Therefore, to build an adversary $\mathcal{B}$ against a pair of real and ideal primitive (small) games, it is enough to decompose the more complex (big) real and ideal games into a reduction $\mathcal{R}$ and the small games. Lemma 2.1 then guarantees if $\mathcal{A}$ can distinguish the big games, then $\mathcal{A} \to \mathcal{R}$ can distinguish the small games.

Lemma 2.1 (Reduction)**.** *Let G^b_{big} and G^b_{small} be two game pairs, $\mathcal{A}$ be an adversary against G^b_{big}, and $\mathcal{R}$ be a (reduction) package such that* $\mathsf{out}(\mathcal{R}) = \mathsf{out}(G^b_{big})$, $\mathsf{in}(\mathcal{R}) = \mathsf{out}(G^b_{small})$ *and $G^b_{big} \overset{code}{\equiv} \mathcal{R} \to G^b_{small}$ for $b \in \{0,1\}$. Let $\mathcal{B} := \mathcal{A} \to \mathcal{R}$. Then, we have*

$$\mathsf{Adv}(\mathcal{A}, G^b_{big}) = \mathsf{Adv}(\mathcal{B}, G^b_{small}).$$

Proof.

$$\begin{aligned}
\mathsf{Adv}(\mathcal{A}, G^b_{big}) &= |\Pr\left[1 = \mathcal{A} \to G^0_{big}\right] - \Pr\left[1 = \mathcal{A} \to G^1_{big}\right]| \\
&= |\Pr\left[1 = \mathcal{A} \to (\mathcal{R} \to G^0_{small})\right] - \Pr\left[1 = \mathcal{A} \to (\mathcal{R} \to G^1_{small})\right]| \\
&=^* |\Pr\left[1 = (\mathcal{A} \to \mathcal{R}) \to G^0_{small}\right] - \Pr\left[1 = (\mathcal{A} \to \mathcal{R}) \to G^1_{small}\right]| \\
&= |\Pr\left[1 = \mathcal{B} \to G^0_{small}\right] - \Pr\left[1 = \mathcal{B} \to G^1_{small}\right]| \\
&= \mathsf{Adv}(\mathcal{B}, G^b_{small})
\end{aligned}$$

where the third equality (marked with star) follows from associativity of sequential composition proved in Lemma 6 of [BDLF+18]. □

Modeling a computational game hop, for an adversary $\mathcal{A}$ and games G^b where $b \in \{0,1\}$, if the advantage $\mathsf{Adv}(\mathcal{A}, G^b)$ is bounded by or equal to another advantage (as a result of reduction to a security assumption), we say games G^0 and G^1 are *computationally equivalent* and write $G^0 \overset{comp}{\approx} G^1$.

2.1.1 Security definitions in SSP

Having defined adversaries and games, we now give indistinguishability against chosen ciphertext attacks (IND-CCA) security definition of a public key encryption scheme Π as an example of a security game in SSP. We will use this definition in Section 2.5. [7]

A public-key encryption (PKE) scheme $\Pi^{\mathsf{PKE}} = (gen, enc, dec)$ consists of three probabilistic algorithms: key generation gen that randomly samples a pair of public-key and secret key (pk, sk); encryption algorithm enc that given message m and public key pk, returns an encryption of m under pk; decryption algorithm dec that given the secret key sk and ciphertext c, returns decryption of c using sk. A PKE scheme

[7] We have adopted the following definitions and KEM-DEM proof from Section 4 of [BDLF+18] with some modifications. For example, in recent papers with SSP, community has used idealization parameters for key packages (keyed packages) for the sake of key idealization instead of idealization parameters for key derivation packages (keying packages). We come back to this topic in Section 2.5.

is required to satisfy the correctness property: for any message m from the message space,

$$\Pr\left[\Pi^{\mathsf{PKE}}.dec(sk, \Pi^{\mathsf{PKE}}.enc(pk, m)) = m;\ (pk, sk) \leftarrow\!\$\ \Pi^{\mathsf{PKE}}.gen()\right] = 1,$$

where the probability is over the randomness of $\Pi^{\mathsf{PKE}}.enc$ and $\Pi^{\mathsf{PKE}}.gen$.

In all games in the rest of the thesis, initial values of all variables and table entries in the state of the games or packages is none or $\perp$. We also denote randomized assignments by $\leftarrow\!\$$ and regular assignments by $\leftarrow$. Moreover, in case of an assertion failure, the game aborts and the adversary notices this outcome.

Definition 2.6 (Public-key IND-CCA security game)**.** Let Π^{PKE} be a PKE scheme. We define PKE-CCA security game $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ as following.

$G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$

Parameters	PKGEN()	PKENC(m)	PKDEC(c')
b : idealization bit	**assert** $sk = \perp$	**assert** $pk \neq \perp$	**assert** $sk \neq \perp$
Π^{PKE} : PKE scheme	$pk, sk \leftarrow\!\$\ \Pi^{\mathsf{PKE}}.gen()$	**assert** $c = \perp$	**assert** $c \neq \perp$
	return pk	**if** $b = 0$ **then**	**assert** $c \neq c'$
State		$c \leftarrow\!\$\ \Pi^{\mathsf{PKE}}.enc(pk, m)$	$m \leftarrow \Pi^{\mathsf{PKE}}.dec(sk, c')$
pk : public key		**else**	**return** m
sk : secret key		$c \leftarrow\!\$\ \Pi^{\mathsf{PKE}}.enc(pk, 0^{\lvert m \rvert})$	
c : challenge		**return** c	

Notice how the game (package) exposes three oracles PKGEN, PKENC, PKDEC to the adversary and make assertions on the state variable $pk \neq \perp$ to enforce the adversary to call PKGEN first and variables $sk = \perp$ and $c = \perp$ respectively to prevent the adversary from generating several key pairs or challenges. In the real game ($b = 0$), the adversary receives an encryption of the message while in the ideal game ($b = 1$) it only receives an encryption of an all-zeros string of the message length.

So far we have only defined security with a pair of real and ideal games and an adversary's advantage when distinguishing these games. However, another game-based definitions involve search-based security notions where the adversary should output a key, group element, a preimage, etc. instead of a single bit. Interestingly, one can model these security notions with a CHECK oracle that returns different values in the real and ideal games if the adversary queries the intended secret. Brzuska and Lipäinen [BL24] explain search games in Appendix B of their work. They also discuss how search games can be modeled as indistinguishability (decision) games. For instance, see the one-way function security game using the CHECK oracle in Section 3.1 of [BL24] or Square and Strong Diffie-Hellman security games (SqDH

and SDH) in Figure 34 of Section 2.5. Also, Kohbrok [Koh23] discusses other SSP security modeling beyond real-vs-ideal paradigm, including search-based notions such as existential unforgeability chosen ciphertext attacks (EUF-CMA), in Section 3.3 of their thesis.

2.2 Paving the way for code equivalence

In the previous section, we mentioned that two games G^l and G^r are said to be code equivalent denoted by $G^l \overset{code}{\equiv} G^r$ if advantage of any adversary $\mathcal{A}$ against them is zero. In this section, we prove the fundamental theorem of code equivalence that gives a general recipe for proving code equivalence of two games.

To show code equivalence, we need to prove any adversary (even unbounded) has zero advantage when distinguishing games G^l and G^r. Therefore, we prove the adversary can not distinguish the outputs from the oracles of the games. Precisely, we show the same oracles from both games return identical outputs if the adversary queries the oracles with identical inputs. Since the oracles are probabilistic, considering probabilistic Turing machines as the computational model, we assume the games choose their random choices from the same randomness tape.

We formalize these ideas as follows and refer to G^l and G^r as left and right games, respectively. We analyze the probability of adversary $\mathcal{A}$ returning 1 against the games G^l and G^r (i.e. $\Pr\left[1 = \mathcal{A} \rightarrow G^l\right]$ and $\Pr[1 = \mathcal{A} \rightarrow G^r]$). For $i \geq 1$, define random variables S_i^l (S_i^r) as the state of game G^l (G^r) after the i-th query, $S_i^{\mathcal{A}\rightarrow G^l}$ ($S_i^{\mathcal{A}\rightarrow G^r}$) as the state of the adversary against G^l (G^r) after the i-th adversary invocation, $O_i^l \in \mathsf{out}(G^l)$ (O_i^r) as the oracle called by the adversary on the i-th query, X_i^l (X_i^r) as the input to oracle O_i^l (O_i^r) chosen by the adversary on the i-th query, Y_i^l (Y_i^r) as the output of oracle O_i^l (O_i^r) on the i-th query, and $B_i^l \in \{\perp, 0, 1\}$ as the output of adversary. We also define Y_0^l and Y_0^r to be $\perp$, i.e. none. As previously mentioned, initial game states S_0^l and S_0^r in SSP assign the value $\perp$ to all variables or table entries. Let $S_0^{\mathcal{A}}$ be a fixed initial state of the adversary. We consider the game oracles and adversaries as state transformers to express relations between the random variables as follows:

$$(Y_i^l, S_i^l) := O_i^l(X_i^l, S_{i-1}^l; R_i^l)$$
$$(O_i^l, X_i^l, S_i^{\mathcal{A}\rightarrow G^l}, B_i^l) := \mathcal{A}(Y_{i-1}^l, S_{i-1}^{\mathcal{A}\rightarrow G^l}; R_i^{\mathcal{A}\rightarrow G^l})$$

The first equation considers the game oracles as the game state transformers and states a relation between the inputs and outputs of the oracles. Given the previous state S_{i-1}^l of the left game after the $(i-1)$-th query and the input X_i^l chosen by the adversary with explicit randomness string R_i^l, the oracle O_i^l computes the next state of the game S_i^l and output Y_i^l that is returned to the adversary. The second equation considers the adversary as its own state transformer. Namely, given its previous state $S_{i-1}^{\mathcal{A}\rightarrow G^l}$ after the first $i-1$ queries and the output Y_{i-1}^l of the oracle O_{i-1}^l on the previous query X_{i-1}^l and explicit randomness string $R_i^{\mathcal{A}\rightarrow G^l}$, the adversary returns its new state $S_i^{\mathcal{A}\rightarrow G^l}$, its chosen oracle name O_i^l, input X_i^l to query the oracle O_i^l on, and output bit B_i^l. If

$B_i^l \neq \perp$, the interaction with the game stops and the adversary returns B_i^l. We consider assertional failures and aborts in oracles as special output values abort that cause the game to stop and disallow any further adversary interaction. Similar equations are defined for the right game with superscript r for all the random variables. Notice that the adversary $\mathcal{A}$ and oracles O_i^l are deterministic with the explicit randomness strings passed as input. In the rest of the section, we might replace l and r with G when discussing both left and right or an arbitrary game G. We want to highlight that any adversary randomness string sequence $R_i^{\mathcal{A}\to G}$ together with a game randomness string sequence R_i^G (for example R_i^l or R_i^r) induce a concrete executation trace and a transcript of interaction between the adversary and the game oracles that ends with an output returned by the adversary.

With this formalization, one can write

$$\Pr\left[1 = \mathcal{A} \to G^l\right] = \sum_{q\geq 1} \Pr\left[1 = \mathcal{A} \to G^l \wedge B_q^l \neq \perp\right] = \sum_{q\geq 1} \Pr\left[B_q^l = 1\right].$$

Observe that event $B_q^l \neq \perp$ occurs if the adversary returns the bit B_q^l on the q-th invokation (i.e. after $q-1$ oracle queries) which prevents any further oracle queries and interaction with the game; therefore, for all q, events $B_q^l \neq \perp$ are all disjoint. We then wish to prove $\Pr\left[B_q^l = 1\right] = \Pr\left[B_q^r = 1\right]$ for all q in order to conclude $\Pr\left[1 = \mathcal{A} \to G^l\right] = \Pr[1 = \mathcal{A} \to G^r]$.

We achieve this result by proving a stronger claim via an induction on q in Lemma 2.3. To prove the induction, though, we need to define state relation I and bijective randomness mapping M. A (game) state relation I is a subset of $S^{G^l} \times S^{G^r}$ where S^{G^l} and S^{G^r} are the set of all possible states of the left and right games, respectively. A pair of game states in a state relation I usually have a meaningful correspondence between them. For example, a relation between the states of two games may state that the public and secret keys are the same. See Definition 2.13 for an example of a state relation between the games $\mathtt{Mod}^{b,0}$ and $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ introduced in the next section. For every oracle $O \in \mathsf{out}(G^l) = \mathsf{out}(G^r)$, randomness mapping $M_O : R_O^{G^l} \longrightarrow R_O^{G^r}$ is a bijective function where $R_O^{G^l}$ and $R_O^{G^r}$ are the set of all possible randomness strings consumed by oracles O^l and O^r, respectively.

Definition 2.7 (Invariant state relation)**.** Let G^l (left) and G^r (right) be two games with the same set of oracles (i.e. $\mathsf{out}(G^l) = \mathsf{out}(G^r)$). For each oracle oracle $O \in \mathsf{out}(G^l)$, let M_O be a randomness mapping between randomness strings consumed by O. Let O^l be the implementation of oracle O in the left game and O^r be the implementation of oracle O in the right game. Denote the initial states of left and right games G^l and G^r by S_0^l and S_0^r, respectively. A state relation I is said to be invariant if the following two conditions hold for all oracles $O \in \mathsf{out}(G^l)$, inputs $x \in \mathsf{inputs}(O)$, states of left

and right games S^l_{old} and S^r_{old}, and randomness string R consumed by O:

$$(S^l_0, S^r_0) \in I \tag{1}$$

$$\left(\begin{array}{c}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}}) \in I \wedge \\ (y^l, S^l_{\mathsf{new}}) \leftarrow O^l(x, S^l_{\mathsf{old}}; R) \wedge \\ (y^r, S^r_{\mathsf{new}}) \leftarrow O^r(x, S^r_{\mathsf{old}}; M_O(R)) \wedge \\ y^l \neq \mathsf{abort} \wedge y^r \neq \mathsf{abort}\end{array}\right) \Longrightarrow (S^l_{\mathsf{new}}, S^r_{\mathsf{new}}) \in I \tag{2}$$

In Lemma 2.2, we justify why such state relations are called invariant. Essentially, for such state relations, initial states of the two games are in relation and upon any oracle query x, states of the games continue to be in relation. Looking forward, this property is at the core of proving code equivalence of left and right games because the states of such games have a specific correspondence that we describe as a state relation and prove to be invariant. For instance, public and secret keys in the left game should be the same as the public and secret keys in the right game. However, notice that the randomness strings consumed by the left and right oracles shall be related via the bijective mapping and can not be arbitrary. We highlight the importance of bijection in the proof of Theorem 2.3.

Next, we introduce the concept of an execution trace. In particular, for fixed adversary randomness and fixed randomness of the left game, we obtain a sequence of oracle inputs and outputs. We want to argue that this sequence stays identical when running the right game on the same sequence of adversarial queries as well as mapped game randomness. Towards this goal, we define the concept of a left-game-induced execution trace which is an execution trace obtained by (a) keeping the adversarial queries induced by the execution trace of the left game, but (b) replacing the oracle outputs of the left game by oracle outputs of the right game (using mapped randomness).

Definition 2.8 (Left-game-induced sequence)**.** Let S^l_0 and S^r_0 be the initial states of left and right games G^l and G^r, $S^{\mathcal{A}}_0$ be the initial state of adversary, $\{R^{\mathcal{A}\to G^l}_i\}$ and $\{R^l_i\}$ be fixed randomness string sequences, and for each oracle $O \in \mathsf{out}(G^l)$, M_O be a randomness mapping. Then the left-game-induced state (oracle, input, output) sequence $S^{l\to r}_q$ $(O^{l\to r}_q, X^{l\to r}_q, Y^{l\to r}_q)$ is defined recursively as follows:

$$(Y^{l\to r}_q, S^{l\to r}_q) := O^{l\to r}_q(X^{l\to r}_q, S^{l\to r}_{q-1}; M_{O^{l\to r}_q}(R^l_q))$$
$$(O^{l\to r}_q, X^{l\to r}_q, S^{\mathcal{A}\to G^r}_q, B^{l\to r}_q) := \mathcal{A}(Y^{l\to r}_{q-1}, S^{\mathcal{A}\to G^r}_{q-1}; R^{\mathcal{A}\to G^l}_q)$$

where $Y^{l\to r}_0 := \perp$ and $S^{l\to r}_q := S^r_0$ and $S^{\mathcal{A}\to G^r}_0 := S^{\mathcal{A}}_0$. We might also refer to the right game randomness sequence defined as $R^r_q := M_{O^{l\to r}_q}(R^l_q)$.

Definition 2.9 (Same-output property)**.** Let I be a state relation between states of the left and right games G^l and G^r. Oracles of G^l and G^r are said to satisfy same-output property if for for all oracles $O \in \mathsf{out}(G^l)$, inputs $x \in \mathsf{inputs}(O)$, states of left and

right games S^l_{old} and S^r_{old}, and randomness string R consumed by O,

$$\left(\begin{array}{c}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}}) \in I \wedge \\ (y^l, S^l_{\mathsf{new}}) \leftarrow O^l(x, S^l_{\mathsf{old}}; R) \wedge \\ (y^r, S^r_{\mathsf{new}}) \leftarrow O^r(x, S^r_{\mathsf{old}}; M_O(R))\end{array}\right) \Longrightarrow y^l = y^r.$$

Lemma 2.2. *Let S^l_0 and S^r_0 be the initial states of left and right games G^l and G^r whose oracles satisfy the same-output property, $S^{\mathcal{A}}_0$ be the initial state of adversary, $\{R^{\mathcal{A}}_i\}$ and $\{R^l_i\}$ be fixed randomness string sequences, I be an invariant state relation, and for each oracle $O \in \mathsf{out}(G^l)$, M_O be a randomness mapping. Then, for all $q \geq 1$, $(S^l_q, S^{l\to r}_q) \in I$, $X^l_q = X^{l\to r}_q$, $Y^l_q = Y^{l\to r}_q$, $B^l_q = B^{l\to r}_q$, $O^l_q = O^{l\to r}_q$, and $S^{\mathcal{A}\to G^l}_q = S^{\mathcal{A}\to G^r}_q$ where $Y^{l\to r}_q$, $B^{l\to r}_q$, $X^{l\to r}_q$, $O^{l\to r}_q$, and $S^{\mathcal{A}\to G^r}_q$ are left-game-induced sequences.*

Proof. We prove via an induction on q. The base case $(S^l_0, S^{l\to r}_0) \in I$ and $Y^l_0 = Y^{l\to r}_0 = \perp$ and $S^{\mathcal{A}\to G^l}_0 = S^{\mathcal{A}\to G^r}_0 = S^{\mathcal{A}}_0$ are clear from definitions. Assume the induction hypothesis holds for $q-1$ where $q \geq 1$. We prove the induction step for q. By definition, $(O^{l\to r}_q, X^{l\to r}_q, S^{\mathcal{A}\to G^r}_q, B^{l\to r}_q) := \mathcal{A}(Y^{l\to r}_{q-1}, S^{\mathcal{A}\to G^r}_{q-1}; R^{\mathcal{A}\to G^l}_q)$ and $(O^l_q, X^l_q, S^{\mathcal{A}\to G^l}_q, B^l_q) := \mathcal{A}(Y^l_{q-1}, S^{\mathcal{A}\to G^l}_{q-1}; R^{\mathcal{A}\to G^l}_q)$. Applying the induction hypothesis that $Y^l_{q-1} = Y^{l\to r}_{q-1}$ and $S^{\mathcal{A}\to G^l}_{q-1} = S^{\mathcal{A}\to G^r}_{q-1}$, we conclude $X^l_q = X^{l\to r}_q$, $B^l_q = B^{l\to r}_q$, $O^l_q = O^{l\to r}_q$, $S^{\mathcal{A}\to G^l}_q = S^{\mathcal{A}\to G^r}_q$. Again, by definition, $(Y^{l\to r}_q, S^{l\to r}_q) := O^{l\to r}_q(X^{l\to r}_q, S^{l\to r}_{q-1}; M_{O^{l\to r}_q}(R^l_q))$ and $(Y^l_q, S^l_q) := O^l_q(X^l_q, S^l_{q-1}; R^l_q)$. Having proved $O^l_q = O^{l\to r}_q$ and $X^l_q = X^{l\to r}_q$ and by induction hypothesis $(S^l_{q-1}, S^{l\to r}_{q-1}) \in I$, we conclude by the invariance of I that $(S^l_q, S^{l\to r}_q) \in I$ and by the same-output property that $Y^l_q = Y^{l\to r}_q$. □

Theorem 2.3 (Fundamental theorem of code equivalence)**.** *Let I be an invariant state relation between states of the left and right games G^l and G^r. If the oracles of the games satisfy same-output property, then, for all $q \geq 1$, oracles $O \in \mathsf{out}(G^l)$, inputs $x \in \mathsf{inputs}(O)$, adversary state $S^{\mathcal{A}}$ and adversary output $b \in \{\perp, 0, 1\}$,*

$$\Pr\left[(O^l_q, X^l_q, S^{\mathcal{A}\to G^l}_q, B^l_q) = (O, x, S^{\mathcal{A}}, b)\right] = \Pr\left[(O^r_q, X^r_q, S^{\mathcal{A}\to G^r}_q, B^r_q) = (O, x, S^{\mathcal{A}}, b)\right],$$

where probabilities are over the choice of $R^{\mathcal{A}\to G^l}_i$'s, $R^{\mathcal{A}\to G^r}_i$'s, R^l_i's and R^r_i's.

Proof. We prove via an induction on q. Recall that $Y^l_0 = Y^r_0 = \perp$. For base case $q = 1$, observe that $(O^l_1, X^l_1, S^{\mathcal{A}\to G^l}_1, B^l_1) = \mathcal{A}(\perp, S^{\mathcal{A}}_0; R^{\mathcal{A}\to G^l}_1)$. Hence, both probabilities are equal to $\Pr\left[\mathcal{A}(\perp, S^{\mathcal{A}}_0; R^{\mathcal{A}\to G^l}_1) = (O, x, S^{\mathcal{A}}, b)\right]$ (over choice of $R^{\mathcal{A}}_1$). Assume the induction hypothesis holds for $q-1$ where $q > 1$. We prove the induction step for q. Again, observe that by definition, $(O^l_q, X^l_q, S^{\mathcal{A}\to G^l}_q, B^l_q) := \mathcal{A}(Y^l_{q-1}, S^{\mathcal{A}\to G^l}_{q-1}; R^{\mathcal{A}\to G^l}_q)$ where $Y^l_{q-1} := O^l_{q-1}(X^l_{q-1}, S^l_{q-2}; R^l_{q-1})$. (We abuse the notation as the oracle returns

(Y^l_{q-1}, S^l_{q-1}).) By conditioning on the triples $(O_{\mathsf{old}}, x', S^{\mathcal{A}}_{\mathsf{old}})$ and applying the law of total probability, we have:

$$\begin{aligned}
&\Pr\left[\mathcal{A}(O^l_{q-1}(X^l_{q-1}, S^l_{q-2}; R^l_{q-1}), S^{\mathcal{A}}_{q-1}; R^{\mathcal{A}\to G^l}_q) = (O, x, S^{\mathcal{A}}, b)\right] \\
&= \sum_{(O_{\mathsf{old}}, x', S^{\mathcal{A}}_{\mathsf{old}})} \Big(\Pr\left[\mathcal{A}(O^l_{\mathsf{old}}(x', S^l_{q-2}; R^l_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^l}_q) = (O, x, S^{\mathcal{A}}, b)\right]\times \\
&\qquad \Pr\left[(O^l_{q-1}, X^l_{q-1}, S^{\mathcal{A}}_{q-1}, B^l_{q-1}) = (O_{\mathsf{old}}, x', S^{\mathcal{A}}_{\mathsf{old}}, \bot)\right]\Big)
\end{aligned}$$

Let $\omega = (O, x, S^{\mathcal{A}}, b)$ and $\omega' = (O_{\mathsf{old}}, x', S^{\mathcal{A}}_{\mathsf{old}}, \bot)$. By applying induction hypothesis to $\Pr\left[(O^l_{q-1}, X^l_{q-1}, S^{\mathcal{A}}_{q-1}, B^l_{q-1}) = \omega'\right]$, it suffices to show for every triple $(O_{\mathsf{old}}, x', S^{\mathcal{A}}_{\mathsf{old}})$ that

$$\Pr\left[\mathcal{A}(O^l_{\mathsf{old}}(x', S^l_{q-2}; R^l_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^l}_q) = \omega\right] = \Pr\left[\mathcal{A}(O^r_{\mathsf{old}}(x', S^r_{q-2}; R^r_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^r}_q) = \omega\right].$$

However,

$$\begin{aligned}
&\Pr\left[\mathcal{A}(O^l_{\mathsf{old}}(x', S^l_{q-2}; R^l_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^l}_q) = \omega\right] \\
&= \sum_{r^l_1, r^l_2, \ldots, r^l_{q-1}} \Big(\Pr\left[\mathcal{A}(O^l_{\mathsf{old}}(x', S^l_{q-2}; R^l_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^l}_q) = \omega | \forall 1 \le i \le q-1.\ R^l_i = r^l_i\right]\times \\
&\qquad \Pr\left[R^l_1 = r^l_1 \wedge \cdots \wedge R^l_{q-1} = r^l_{q-1}\right]\Big) \\
&= \sum_{r^l_1, r^l_2, \ldots, r^l_{q-1}} \Big(\Pr\left[\mathcal{A}(O^r_{\mathsf{old}}(x', S^{l\to r}_{q-2}; R^r_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^r}_q) = \omega | \forall 1 \le i \le q-1.\ R^r_i = M_{O_i}(r^l_i)\right]\times \\
&\qquad \Pr\left[R^l_1 = r^l_1 \wedge \cdots \wedge R^l_{q-1} = r^l_{q-1}\right]\Big) \\
&= \sum_{r^l_1, r^l_2, \ldots, r^l_{q-1}} \Big(\Pr\left[\mathcal{A}(O^r_{\mathsf{old}}(x', S^{l\to r}_{q-2}; R^r_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^r}_q) = \omega | \forall 1 \le i \le q-1.\ R^r_i = M_{O_i}(r^l_i)\right]\times \\
&\qquad \Pr\left[R^r_1 = M_{O_1}(r^l_1) \wedge \cdots \wedge R^r_{q-1} = M_{O_1}(r^l_{q-1})\right]\Big) \\
&= \sum_{r^r_1, r^r_2, \ldots, r^r_{q-1}} \Big(\Pr\left[\mathcal{A}(O^r_{\mathsf{old}}(x', S^r_{q-2}; R^r_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^r}_q) = \omega | \forall 1 \le i \le q-1.\ R^r_i = r^r_i\right]\times \\
&\qquad \Pr\left[R^r_1 = r^r_1 \wedge \cdots \wedge R^r_{q-1} = r^r_{q-1}\right]\Big) \\
&= \Pr\left[\mathcal{A}(O^r_{\mathsf{old}}(x', S^r_{q-2}; R^r_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^r}_q) = \omega\right].
\end{aligned}$$

The second equality follows from Lemma 2.2 and the same-output property. Observe that Lemma 2.2 shows $(S^l_{q-2}, S^{l\to r}_{q-2}) \in I$, $R^r_i = M_{O_i}(r^l_i)$, $O^{l\to r}_{q-1} = O^l_{q-1} = O_{\mathsf{old}}$,

$X^{l\to r}_{q-1} = X^{l}_{q-1} = x'$, and $S^{\mathcal{A}}_{\mathsf{old}} = S^{\mathcal{A}\to G^l}_{q-1} = S^{\mathcal{A}\to G^r}_{q-1}$. Therefore, applying the same-output property yields $O^{l}_{\mathsf{old}}(x', S^{l}_{q-2}; R^{l}_{q-1}) = O^{r}_{\mathsf{old}}(x', S^{l\to r}_{q-2}; R^{r}_{q-1})$ and proves the following probabilities over the choice of $R^{\mathcal{A}}_i$'s are equal: [8]

$$\Pr\Big[\mathcal{A}(O^{l}_{\mathsf{old}}(x', S^{l}_{q-2}; R^{l}_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^l}_{q}) = \omega | \forall 1 \le i \le q-1.\ R^{l}_{i} = r^{l}_{i}\Big]$$
$$= \Pr\Big[\mathcal{A}(O^{r}_{\mathsf{old}}(x', S^{l\to r}_{q-2}; R^{r}_{q-1}), S^{\mathcal{A}}_{\mathsf{old}}; R^{\mathcal{A}\to G^r}_{q}) = \omega | \forall 1 \le i \le q-1.\ R^{r}_{i} = M_{O_i}(r^{l}_{i})\Big].$$

The third equality follows from bijection of randomness mappings M_{O_i}'s, independence of R^{l}_{i}'s and R^{r}_{i}'s, and their uniform distributions. Last equality follows from renaming $M_{O_i}(r^{l}_{i})$ to r^{r}_{i} as a result of bijection of randomness mappings M_{O_i}'s. Notice that after renaming $S^{l\to r}_{q-2}$ is exactly S^{r}_{q-2}. □

Corollary 2.3.1. *Let I be an invariant state relation between states of the left and right games G^l and G^r. If the oracles of the games satisfy same-output property, then, for all $q \ge 1$, $\Pr\big[B^{l}_{q} = 1\big] = \Pr\big[B^{r}_{q} = 1\big]$.*

2.3 CCA-Secuirty of KEM-DEM

Next, we illustrate an example of a security reduction in SSP by revisiting KEM-DEM paradigm for building public-key encryption (PKE) schemes indistinguishable under chosen-ciphertext attacks (IND-CCA). KEM-DEM is a popular example for illustrating proofs in the SSP framework. For example, the original paper on SSP [BDLF+18] and the formalization effort of SSP in EasyCrypt [DKO21] both have used the KEM-DEM example.

Key encapsulation mechanism (KEM) is a public-key cryptographic primitive that allows to encapsulate a (short) symmetric key using a public key. The symmetric key then allows to encrypt a (long) message with a symmetric encryption scheme. When combined with KEM, the symmetric encryption scheme is referred to as Data encapsulation mechanism (DEM). Initially, a public and secret key pair (pk, sk) is generated by the key generation algorithm of the KEM scheme. To encrypt a (long) message m with public key pk, a (short) fresh symmetric key k is generated and encapsulated (encrypted) as c_k with key encapsulation algorithm of the KEM scheme using pk. In the next step, m is encrypted to c_m with the encryption algorithm of DEM scheme using k. The final cipher text will then be (c_k, c_m). To decrypt a ciphertext (c_k, c_m) using secret key sk, c_k is decapsulated using sk to obtain k. Then, c_m is decrypted with the decryption algorithm of DEM scheme using k.

Therefore, KEM-DEM is a generic construction of a hybrid public-key encryption scheme using a KEM and DEM primitive. Such hybrid public-key encryption constructions enjoy fast encryption despite computational-heavy public-key encryption by delegating the hard work of encrypting long messages to fast symmetric key encryption schemes.

[8] One can prove this in more details by conditioning on concrete string choices for $R^{\mathcal{A}}_i$'s. After conditioning probabilities will be exactly equal (both zero or one) because of inputs of $\mathcal{A}$ are exactly equal.

In the following, we formally define the syntax of a KEM scheme Π^{KEM} and a DEM scheme Π^{DEM} as well as a candidate KEM-DEM hybrid construction Π^{PKE}. We then define KEM-CCA and DEM-CCA security for Π^{KEM} and Π^{DEM}, respectively. Finally, we reduce PKE-CCA security of Π^{PKE} to KEM-CCA and DEM-CCA with a game hopping hybrid argument.

Formally, a KEM scheme $\Pi^{\ell,\mathsf{KEM}} = (gen, encaps, decaps)$ consists of three probabilistic algorithms: key generation, which returns a pair (pk, sk) of public key and secret key; encapsulation, which given pk returns a key k of length ℓ and encapsulation c_k of k; decapsulation, which given sk and a an encapsulated key c_k returns raw key k (decapsulation of c_k). A KEM scheme is required to satisfy the correctness property: for any message m from the message space,

$$\Pr\left[\Pi^{\ell,\mathsf{KEM}}.decaps(sk, c_k) = k;\ \begin{array}{c}(c_k, k) \leftarrow\$\ \Pi^{\ell,\mathsf{KEM}}.encaps(pk)\\ (pk, sk) \leftarrow\$\ \Pi^{\ell,\mathsf{KEM}}.gen()\end{array}\right] = 1,$$

where the probability is over the randomness of $\Pi^{\mathsf{KEM}}.encaps$ and $\Pi^{\mathsf{KEM}}.gen$.

A DEM scheme $\Pi^{\ell',\mathsf{DEM}} = (enc, dec)$ consists of two probabilistic algorithms: encryption and decryption algorithms that both require a key of length ℓ' provided by the KEM scheme. (Notice the lack of key generation algorithm compared to the symmetric key encryption schemes.) A DEM scheme is required to satisfy the correctness property: for any message m from the message space,

$$\Pr\left[\Pi^{\ell',\mathsf{DEM}}.dec(k, \Pi^{\ell',\mathsf{DEM}}.enc(k, m)) = m;\ k \leftarrow\$\ \{0,1\}^{\ell'}\right] = 1,$$

where the probability is over the randomness of $\Pi^{\ell',\mathsf{DEM}}.enc$.

We define the following candidate hybrid construction Π^{PKE} of a PKE scheme based on $\Pi^{\ell,\mathsf{KEM}}$ and $\Pi^{\ell',\mathsf{DEM}}$ where $\ell = \ell' = \lambda$.

Definition 2.10 (Candidate PKE construction)**.** Let $\Pi^{\lambda,\mathsf{KEM}}$ and $\Pi^{\lambda,\mathsf{DEM}}$ be respectively KEM and DEM schemes with key length λ. We define PKE scheme Π^{PKE} as follows:

$\Pi^{\mathsf{PKE}}.gen()$	$\Pi^{\mathsf{PKE}}.enc(pk, m)$	$\Pi^{\mathsf{PKE}}.dec(sk, c)$
return $\Pi^{\lambda,\mathsf{KEM}}.gen()$	$c_k, k \leftarrow\$\ \Pi^{\lambda,\mathsf{KEM}}.encaps(pk)$	$(c_k, c_m) \leftarrow$ **parse** c
	$c_m \leftarrow\$\ \Pi^{\lambda,\mathsf{DEM}}.enc(k, m)$	$k \leftarrow \Pi^{\lambda,\mathsf{KEM}}.decaps(sk, c_k)$
	return (c_k, c_m)	$m \leftarrow \Pi^{\lambda,\mathsf{DEM}}.dec(k, c_m)$
		return m

Hereafter, for simplicity, we denote $\Pi^{\lambda,\mathsf{KEM}}$ and $\Pi^{\lambda,\mathsf{DEM}}$ by Π^{KEM} and Π^{DEM}.

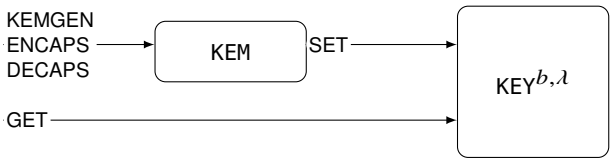
We reduce the PKE-CCA security of construction Π^{PKE} to KEM-CCA and DEM-CCA security of schemes $\Pi^{\lambda,\mathsf{KEM}}$ and $\Pi^{\lambda,\mathsf{DEM}}$, respectively, a result first shown by Cramer and Shoup [CS98]. Therefore, we first define KEM-CCA and DEM-CCA security notions in SSP and in the next step prove the reduction in Theorem 2.4.

Figure 4 illustrates the definition of packages KEM, DEM, and KEY. The KEY package is a common paradigm in security modeling with SSP for sharing state (keys) between several packages. Usually, one of these packages (keying packages), performing key

derivations or key generation, sets a key in the package KEY and one or more packages (keyed packages) consume the key in the package. In KEM-DEM hybrid construction, the KEM package generates a symmetric key and sets it in the KEY package from which the DEM package retrieves the key for encryption. We will see another example of key storage packages in TLS security games in Section 3.2.

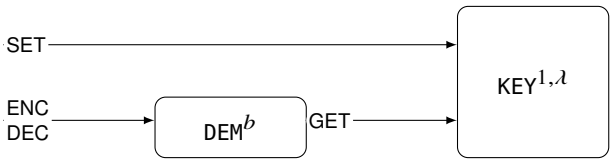
Notice the $\mathsf{KEY}^{b,\lambda}$ package is parameterized with b and λ. Upon a SET oracle call to the ideal package ($b = 1$), a uniformly random key of length λ is sampled and set in the package, regardless of the oracle input. However, in the real package ($b = 0$), the key provided as an input to the oracle is set in the package. We use the ideal key package for $G^{b,\Pi^{\lambda,\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}}$ to allow the adversary to instruct the game with a random key, while in $G^{b,\Pi^{\lambda,\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}}$, we require the adversary to distinguish between a random key and the real key even if it observes the encapsulation of the key.

Definition 2.11 (KEM-CCA security game)**.** Let KEM and KEY be the packages defined in Figure 4. We define the KEM-CCA security game $G^{b,\Pi^{\lambda,\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}} := \frac{\mathsf{KEM}}{\mathsf{ID}_{\mathsf{GET}}} \rightarrow \mathsf{KEY}^{b,\lambda}$ visualized below. Notice the adversary has access to four oracles {KEMGEN, ENCAPS, DECAPS, GET}.



Similar to PKE-CCA game, $G^{b,\Pi^{\lambda,\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}}$ uses assertions to force the adversary to call KEMGEN oracle and generate public and secret keys pair before calling other oracles. In the real game the adversary observes the real key (via GET), its encapsulation (via ENCAPS), and decapsulation of any non challenge ciphertext (via DECAPS). In the ideal game the adversary receives a random key instead of the real key because the input to SET oracle is ignored. However, the adversary still receives the encapsulation of real key.

Definition 2.12 (DEM-CCA security game)**.** Let DEM and KEY be the packages defined in Figure 4. We define the DEM-CCA security game $G^{b,\Pi^{\lambda,\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}} := \frac{\mathsf{ID}_{\mathsf{SET}}}{\mathsf{DEM}^b} \rightarrow \mathsf{KEY}^{1,\lambda}$, visualized below. Notice the adversary has access to three oracles {SET, ENC, DEC}.



Similar to KEM-CCA security game, $G^{b,\Pi^{\lambda,\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}}$ requires the adversary to call the SET oracle so a random key is generated by the game. In the real game, upon calling the oracle ENC, the adversary receives an encryption of the real message as a challenge while in the ideal game, the adversary receives an encryption of all-zeros string. In both games, the adversary can request decryption of non-challenge (those not queried to ENC) ciphertexts. The game uses a set S to record the challenge ciphertexts in oracle ENC.

KEM

Parameters

Π^{KEM} : KEM scheme

State

pk : public key

sk : secret key

c : challenge

KEMGEN()

assert $sk = \bot$

$pk, sk \leftarrow\$ \Pi^{\mathsf{KEM}}.gen()$

return pk

ENCAPS()

assert $pk \neq \bot$

assert $c = \bot$

$c, k \leftarrow\$ \Pi^{\mathsf{KEM}}.encaps(pk)$

SET(k)

return c

DECAPS(c')

assert $sk \neq \bot$

assert $c \neq c'$

$k \leftarrow\$ \Pi^{\mathsf{KEM}}.decaps(sk, c')$

return k

DEM^b

Parameters

b : idealization bit

Π^{DEM} : DEM scheme

State

S : set

ENC(m)

$k \leftarrow$ GET()

if $b = 1$:

  $c \leftarrow\$ \Pi^{\mathsf{DEM}}.enc(k, 0^{|m|})$

else :

  $c \leftarrow\$ \Pi^{\mathsf{DEM}}.enc(k, m)$

$S \leftarrow S \cup \{c\}$

return c

DEC(c)

assert $c \notin S$

$k \leftarrow$ GET()

$m \leftarrow \Pi^{\mathsf{DEM}}.dec(k, c)$

return m

$\mathsf{KEY}^{b,\lambda}$

Parameters

b : idealization bit

λ : integer

State

k : key

SET(k')

assert $k = \bot$

if $b = 1$:

  $k \leftarrow\$ \{0,1\}^{\lambda}$

else :

  $k \leftarrow k'$

GET()

assert $k \neq \bot$

return k

Comb

Parameters

Π^{DEM} : DEM scheme

State

pk : public key

c : challenge

PKGEN()

assert $pk = \bot$

$pk \leftarrow\$$ KEMGEN()

return pk

PKENC(m)

assert $pk \neq \bot$

assert $c = \bot$

$c_k \leftarrow\$$ ENCAPS()

$c_m \leftarrow\$$ ENC(m)

$c \leftarrow (c_k, c_m)$

return c

PKDEC(c')

assert $pk \neq \bot$

assert $c \neq c'$

$(c_k, c_m) \leftarrow$ **parse** c

$(c'_k, c'_m) \leftarrow$ **parse** c'

if $c'_k = c_k$:

  $m \leftarrow$ DEC(c'_m)

else :

  $k' \leftarrow$ DECAPS(c'_k)

  $m \leftarrow \Pi^{\mathsf{DEM}}.dec(k', c'_m)$

return m

Figure 4: Packages KEM, DEM, KEY, and Comb

Theorem 2.4. *Let $\Pi^{\lambda,\mathit{PKE}}$ be the PKE construction in definition 2.10 and $\mathcal{A}$ be an adversary. There exist probabilistic polynomial-time (PPT) reductions $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ such that*

$$\begin{aligned}\mathsf{Adv}(\mathcal{A}, G^{b,\Pi^{\mathit{PKE}}}_{\mathit{PKE\text{-}CCA}}) \leq\ & \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_1, G^{b,\Pi^{\mathit{KEM}}}_{\mathit{KEM\text{-}CCA}}) \\ &+ \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_2, G^{b,\Pi^{\mathit{DEM}}}_{\mathit{DEM\text{-}CCA}}) \\ &+ \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_3, G^{b,\Pi^{\mathit{KEM}}}_{\mathit{KEM\text{-}CCA}}).\end{aligned}$$

Proof. Let us define the combiner package $\mathtt{Comb}$ in Figure 4 which combines the output interfaces of packages $\mathtt{KEM}$ and $\mathtt{DEM}$ analogous to the construction $\Pi^{\lambda,\mathtt{PKE}}$. Define the parameterized modular game $\mathtt{Mod}^{b_1,b_2} := \mathtt{Comb} \to \frac{\mathtt{KEM}}{\mathtt{DEM}^{b_1}} \to \mathtt{KEY}^{b_2,\lambda}$. Next, we define hybrid games H_i for $i \in \{1,2,3,4\}$ using the generic game $\mathtt{Mod}^{b_1,b_2}$ as follows:

$$\begin{aligned}H_1 &:= \mathtt{Mod}^{0,0} = \mathtt{Comb} \to \frac{\mathtt{KEM}}{\mathtt{DEM}^0} \to \mathtt{KEY}^{0,\lambda} \\ H_2 &:= \mathtt{Mod}^{0,1} = \mathtt{Comb} \to \frac{\mathtt{KEM}}{\mathtt{DEM}^0} \to \mathtt{KEY}^{1,\lambda} \\ H_3 &:= \mathtt{Mod}^{1,1} = \mathtt{Comb} \to \frac{\mathtt{KEM}}{\mathtt{DEM}^1} \to \mathtt{KEY}^{1,\lambda} \\ H_4 &:= \mathtt{Mod}^{1,0} = \mathtt{Comb} \to \frac{\mathtt{KEM}}{\mathtt{DEM}^1} \to \mathtt{KEY}^{0,\lambda}\end{aligned}$$

See visualization of these hybrid games in Figure 5. Briefly, H_1 is the modular variant of $G^{0,\Pi^{\mathtt{PKE}}}_{\mathtt{PKE\text{-}CCA}}$; H_2 is the same as H_1 except that the $\mathtt{KEY}$ package is idealized (i.e. $\mathtt{KEY}^{1,\lambda}$); H_3 is the same as H_2 except that the $\mathtt{DEM}$ package is idealized (i.e. $\mathtt{DEM}^1$); finally, H_4 is the modular variant of $G^{1,\Pi^{\mathtt{PKE}}}_{\mathtt{PKE\text{-}CCA}}$ and when compared to H_3, the $\mathtt{KEY}$ package is deidealized (i.e. $\mathtt{KEY}^{0,\lambda}$).

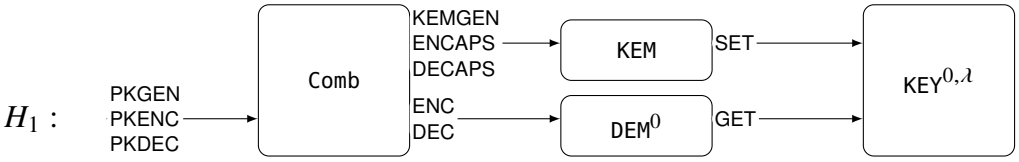
Having defined the hybrid games, we informally show the following game hops:

$$G^{0,\Pi^{\mathtt{PKE}}}_{\mathtt{PKE\text{-}CCA}} \overset{code}{\equiv} H_1 \overset{comp}{\approx} H_2 \overset{comp}{\approx} H_3 \overset{comp}{\approx} H_4 \overset{code}{\equiv} G^{1,\Pi^{\mathtt{PKE}}}_{\mathtt{PKE\text{-}CCA}}$$

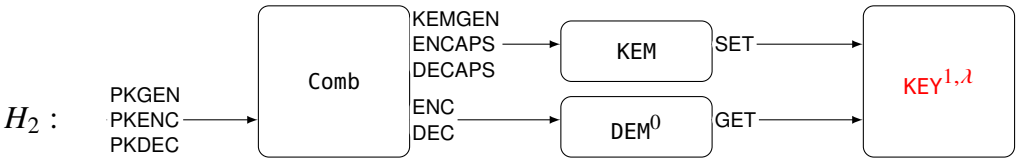
Define reductions $\mathcal{R}_i$ for $i \in \{1,2,3\}$ as follows:

$$\begin{aligned}\mathcal{R}_1 &:= \mathtt{Comb} \to \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{KEM})}}{\mathtt{DEM}^0} \\ \mathcal{R}_2 &:= \mathtt{Comb} \to \frac{\mathtt{KEM}}{\mathtt{ID}_{\mathsf{out}(\mathtt{DEM}^b)}} \\ \mathcal{R}_3 &:= \mathtt{Comb} \to \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{KEM})}}{\mathtt{DEM}^1}\end{aligned}$$

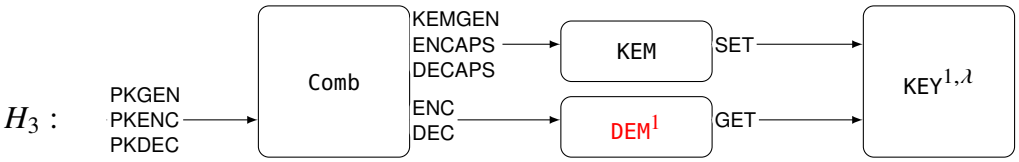
See Figure 6 for visualization of reductions (red packages) as graph cuts in the call graph of the hybrid games H_i. Notice that we do not show identity packages in the graph cuts.



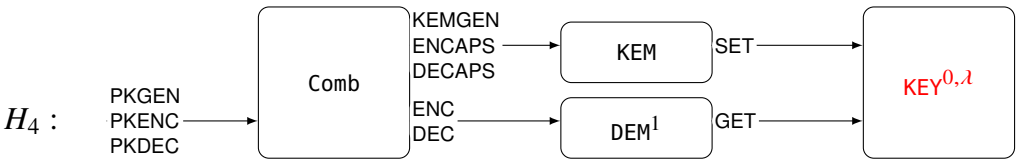
(a) H_1



(b) H_2

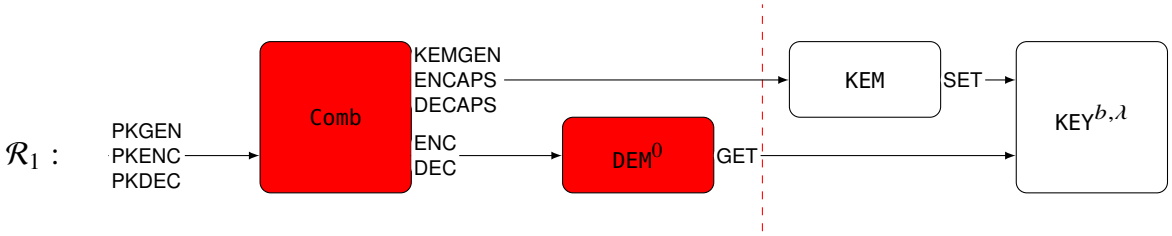


(c) H_3

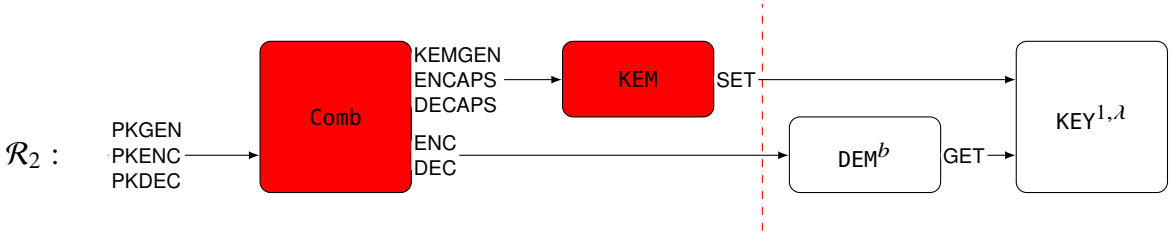


(d) H_4

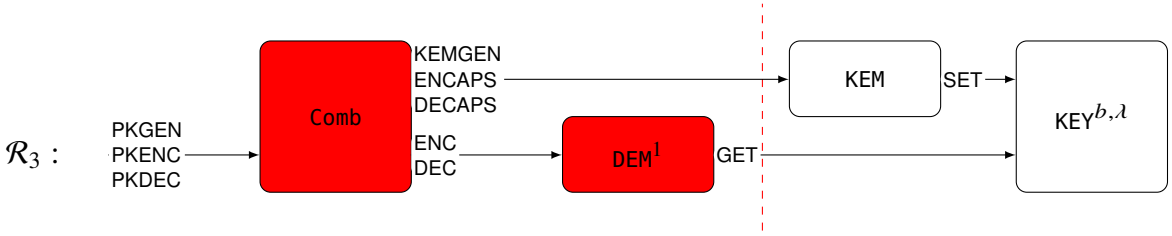
Figure 5: Hybrid games H_i for $i \in \{1, 2, 3, 4\}$



(a) $\mathcal{R}_1$



(b) $\mathcal{R}_2$



(c) $\mathcal{R}_3$

Figure 6: Reductions $\mathcal{R}_i$ for $i \in \{1, 2, 3\}$ as red packages

Formally, we prove the following code equivalences in Claims 2.6 and 2.5:

$$H_1 \overset{code}{\equiv} G^{0,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}},\ H_4 \overset{code}{\equiv} G^{1,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}} \qquad \text{(Claim 2.6)}$$

$$H_1 \overset{code}{\equiv} \mathcal{R}_1 \to G^{0,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}},\ H_2 \overset{code}{\equiv} \mathcal{R}_1 \to G^{1,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}} \qquad \text{(Claim 2.5)}$$

$$H_2 \overset{code}{\equiv} \mathcal{R}_2 \to G^{0,\Pi^{\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}},\ H_3 \overset{code}{\equiv} \mathcal{R}_2 \to G^{1,\Pi^{\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}} \qquad \text{(Claim 2.5)}$$

$$H_3 \overset{code}{\equiv} \mathcal{R}_3 \to G^{1,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}},\ H_4 \overset{code}{\equiv} \mathcal{R}_3 \to G^{0,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}} \qquad \text{(Claim 2.5)}$$

Combining all,

$$\begin{aligned}
\mathsf{Adv}(\mathcal{A}, G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}) &= |\Pr\left[1 = \mathcal{A} \to G^{0,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}\right] - \Pr\left[1 = \mathcal{A} \to G^{1,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}\right]| \\
&= |\Pr[1 = \mathcal{A} \to H_1] - \Pr[1 = \mathcal{A} \to H_4]| \\
&= |\Pr[1 = \mathcal{A} \to H_1] - \Pr[1 = \mathcal{A} \to H_2] \\
&\quad + \Pr[1 = \mathcal{A} \to H_2] - \Pr[1 = \mathcal{A} \to H_3] \\
&\quad + \Pr[1 = \mathcal{A} \to H_3] - \Pr[1 = \mathcal{A} \to H_4]| \\
&\leq |\Pr[1 = \mathcal{A} \to H_1] - \Pr[1 = \mathcal{A} \to H_2]| \\
&\quad + |\Pr[1 = \mathcal{A} \to H_2] - \Pr[1 = \mathcal{A} \to H_3]| \\
&\quad + |\Pr[1 = \mathcal{A} \to H_3] - \Pr[1 = \mathcal{A} \to H_4]| \\
&= \mathsf{Adv}(\mathcal{A}, H_1, H_2) \\
&\quad + \mathsf{Adv}(\mathcal{A}, H_2, H_3) \\
&\quad + \mathsf{Adv}(\mathcal{A}, H_3, H_4) \\
&= \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_1, G^{b,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}}) \\
&\quad + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_2, G^{b,\Pi^{\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}}) \\
&\quad + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_3, G^{b,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}}),
\end{aligned}$$

where the second equality follows from Claim 2.6, the second to the last inequality follows by triangle inequality and last equality follows from Claim 2.5 and Lemma 2.1 applied to big games $G^0_{i,big} := H_i$ and $G^1_{i,big} := H_{i+1}$ for $i \in \{1, 2, 3\}$ (i.e. game pairs (H_1, H_2), (H_2, H_3), and (H_4, H_4)) and reductions $\mathcal{R}_i$. □

Claim 2.5. *For $b \in \{0, 1\}$, $H_{1+b} \overset{code}{\equiv} \mathcal{R}_1 \to G^{b,\Pi^{KEM}}_{KEM\text{-}CCA}$, $H_{2+b} \overset{code}{\equiv} \mathcal{R}_2 \to G^{b,\Pi^{DEM}}_{DEM\text{-}CCA}$, and $H_{3+b} \overset{code}{\equiv} \mathcal{R}_3 \to G^{b,\Pi^{KEM}}_{KEM\text{-}CCA}$.*

Proof. The code equivalences can be verified by inlining the code of small games $G^{b,\Pi^{\mathsf{DEM}}}_{\mathsf{DEM\text{-}CCA}}$ and $G^{b,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}}$ in the corresponding reductions $\mathcal{R}_i$ and comparing the resulting pseudocode with the inlined code of games H_i, which are exactly the same. The other approach is to verify the code equivalence with algebraic notation of package composition. For instance, since H_1 is defined as $\mathtt{Comb} \to \frac{\mathsf{KEM}}{\mathsf{DEM}^0} \to \mathsf{KEY}^{0,\lambda}$, we have

$$\mathcal{R}_1 \to G^{0,\Pi^{\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}} \overset{code}{\equiv} \left(\mathtt{Comb} \to \frac{\mathtt{ID}_{\mathsf{out}(\mathsf{KEM})}}{\mathsf{DEM}^0}\right) \to \left(\frac{\mathsf{KEM}}{\mathtt{ID}_{\mathsf{GET}}} \to \mathsf{KEY}^{0,\lambda}\right) \overset{code}{\equiv} \mathtt{Comb} \to \frac{\mathsf{KEM}}{\mathsf{DEM}^0} \to \mathsf{KEY}^{0,\lambda}.$$

□

Claim 2.6. *$H_1 \overset{code}{\equiv} G^{0,\Pi^{PKE}}_{PKE\text{-}CCA}$ and $H_4 \overset{code}{\equiv} G^{1,\Pi^{PKE}}_{PKE\text{-}CCA}$.*

Proof. Notice $\mathtt{Mod}^{b,0} := \mathtt{Comb} \to \frac{\mathsf{KEM}}{\mathsf{DEM}^b} \to \mathsf{KEY}^{0,\lambda}$. We prove $\mathtt{Mod}^{b,0} \overset{code}{\equiv} G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ for $b \in \{0, 1\}$ which implies the required equivalences since $\mathtt{Mod}^{0,0} = H_1$ and $\mathtt{Mod}^{1,0} = H_4$. We apply the Fundamental Theorem of Code Equivalence (Theorem 2.3) and specifcially Corollary 2.3.1 from previous section to prove the code equivalence. We first present a state relation I for the games $\mathtt{Mod}^{b,0}$ and $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$.

Definition 2.13 (State relation for states of $\mathsf{Mod}^{b,0}$ and $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$)**.** For states S^l and S^r, $(S^l, S^r) \in I$ if

$$pk_{\mathsf{left}} = pk_{\mathsf{right}} \land sk_{\mathsf{left}} = sk_{\mathsf{right}} \tag{1}$$
$$pk_{\mathsf{left}} = \bot \Leftrightarrow sk_{\mathsf{left}} = \bot \tag{2}$$
$$c^{\mathsf{PKE}}_{\mathsf{left}} = c_{\mathsf{right}} \tag{3}$$
$$c^{\mathsf{PKE}}_{\mathsf{left}} = \bot \Leftrightarrow k_{\mathsf{left}} = \bot \Leftrightarrow c^{\mathsf{KEM}}_{\mathsf{left}} = \bot \tag{4}$$
$$\left(c^{\mathsf{PKE}}_{\mathsf{left}} = \bot \Rightarrow S = \varnothing\right) \land \left(c^{\mathsf{PKE}}_{\mathsf{left}} \neq \bot \Rightarrow (S = \{c_m\} \land (\star, c_m) = c^{\mathsf{PKE}}_{\mathsf{left}})\right) \tag{5}$$
$$k_{\mathsf{left}} \neq \bot \Rightarrow k_{\mathsf{left}} = \Pi^{\mathsf{KEM}}.decaps(sk_{\mathsf{left}}, c^{\mathsf{KEM}}_{\mathsf{left}}) \tag{6}$$
$$c^{\mathsf{PKE}}_{\mathsf{left}} \neq \bot \Rightarrow c^{\mathsf{PKE}}_{\mathsf{left}} = (c^{\mathsf{KEM}}_{\mathsf{left}}, \star) \tag{7}$$

where the left and right subscripts are used to denote the variables in S^l and S^r, respectively.

Observe that state relations 3, 4, 5, 6, and 7 describe properties only for the left game. Intuitively, one should be able to argue their invariance independently from the right game. We refer to these state relations (invariants) as one-sided state relations (invariants). In Section 4.2.5, we present the Invariant bubbling theorem about the one-sided state relations that simplifies proving invariance of such state relations.

Definition 2.14 (One-sided and two-sided state relations)**.** A one-sided state relation is a state relation that expresses a property only about the state of one of the left or right games. A two-sided state relation is a state relations that describes a property between the states of both of the left and the right game.

Definition 2.15 (Randomness mapping for oracles of $\mathsf{Mod}^{b,0}$ and $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$)**.** Although package oracles defined in Figure 4 do not accept an explicit randomness, we can imagine PKGEN receives a randomness string and use it for key generation. Therefore, the mapping can simply be an identity mapping. (i.e. the strings are equal) For PKENC, we can imagine the randomness string is split into two halfs such that each half is consumed by encryption and encapsulation. The mapping can again be simply an identity mapping. Concrelty, we assume $\Pi^{\mathsf{KEM}}.gen$ returns the same key pair (pk, sk) in both games, $\Pi^{\mathsf{KEM}}.encaps$ reutrns the same encapsulated key (c_k, k) on pk, and $\Pi^{\mathsf{DEM}}.enc$ returns the same ciphertext c_m on m.

Notice the bijection of randomness mapping of Definition 2.15 is obvious. In the following claim, we show invariance of I and the same-output property in order to apply Theorem 2.3 and conclude $\Pr\left[1 = \mathcal{A} \rightarrow G^l\right] = \Pr[1 = \mathcal{A} \rightarrow G^r]$ from Corollary 2.3.1. This concludes the proof of Claim 2.6. □

Claim 2.7. *Let $G^l := \mathsf{Mod}^{b,0}$ and $G^r := G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ and I be the state relation in Definition 2.13. Then, I is an invariant state relation and the oracles of games G^l and G^r satisfy the same-output property.*

Proof. We begin by proving the same-output property. Recall that we consider assertion failures and oracle abort as a special output value $\bot$ returned by the oracles

that stop the games. For ease of referencing lines of the game pseudocodes in the following code-based argument, we have inlined the code of all packages $\mathtt{Comb}$, $\mathtt{KEM}$, $\mathtt{DEM}^b$, $\mathtt{KEY}^{0,\lambda}$ recursively in $\mathtt{Mod}^{b,0}$ and inlined the hybrid construction Π^{PKE} in $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ in Figure 7. Red lines, only shown for demonstration, are original oracle calls that are replaced with blue lines indicating inlined code. Therefore, oracle codes consist of only blue and black lines. Notice how the $\mathsf{SET}(k)$ oracle call is also inlined as part of recursive inlining.

For PKGEN, using the invariant 1, left and right secret keys are the same. (both are $\perp$ or have equal values) Therefore, either the assertion fails in both left and right games or neither of the assertions fail. If the assertion does not fail, the oracle returns the same public key and secret key due to the randomness mapping.

For PKENC, using the invariants 1, assertion **assert** $pk \neq \perp$ either fails in both the left and right games or in neither fails. Similarly, using invariant 3, assertions **assert** $c^{\mathsf{PKE}} = \perp$ and **assert** $c = \perp$ either both fail or neither fail. Invariant 4 ensures that other assertions (**assert** $c^{\mathsf{KEM}} = \perp$ and **assert** $k = \perp$) in the left game never fail. Using the randomness mapping for encapsulation and encryption as well as invariant 1 that public keys are equal, it can be seen that oracles return the same encapsulated keys and encrypted messages.

For PKDEC, using invariants 1 and 2, assertions **assert** $pk \neq \perp$ and **assert** $sk \neq \perp$ in the left and right game either both fails or neither fails. Similarly, using invariant 3, assertions **assert** $c \neq c'$ and **assert** $c^{\mathsf{PKE}} \neq c'$ have the same failing status. Moreover, if the **else** branch in the left game is chosen, the same output is returned from both oracle. (Invariant 7 prevents failure of assertion **assert** $c^{\mathsf{KEM}} \neq c'_k$ because $c_k = c^{\mathsf{KEM}}$.) Now, consider the branch $c'_k = c_k$ is chosen. Since $c' \neq c^{\mathsf{PKE}}$, then $c'_m \neq c_m$. Therefore, using invariant 5, assertion **assert** $c'_m \notin S$ does not fail. Finally, invariants 6 and 7 prove that the output of the oracle in left game is the same as right game.

To prove invariance of I, observe that PKDEC does not modify the state. Moreover, it is easy to see that PKGEN preserves the invariants. To prove invariant 6, the correctness property of KEM scheme is required (i.e. $k = \Pi^{\mathsf{KEM}}.decaps(sk, c_k)$ where $(c_k, k) \leftarrow\!\!\$\ \Pi^{\mathsf{KEM}}.encaps(pk)$). To prove invariant 5, observe that only one element is added to S because the assertions in PKENC prevent the adversary from generating several challenges. Other invariants can be easily proven using the randomness mapping and code logic. □

Remark. We want to emphasize that Theorem 2.3 lays the foundation for SSBee that we introduce in Section 2.5. Looking ahead, SSBee helps to automate the code equivalence proofs such as proof of Claim 2.7. It is noteworthy why the automation is very useful as the complexity of code-based argument such as the one in proof of Claim 2.7 can easily get out of control as the security games become more complex. The code equivalence proof technique demonstrated in this section was first introduced by Brzuska, Delignat-Lavaud, Egger, Fournet, Kohbrok, Kohlweiss (BDEFKK) [BDLE+21] in their analysis of TLS 1.3 key schedule security analysis and is extensively used in this thesis. We want to emphasize that the formalization, statement, and proof of the Theorem 2.3 is a contribution of this thesis as a result of hours of discussion with my supervisors. Although Brzuska, Delignat-Lavaud, Egger,

$\mathsf{Mod}^{b,0}$

Parameters

b : idealization bit
Π^{KEM} : KEM scheme
Π^{DEM} : DEM scheme

State

pk : public key
sk : secret key
c^{PKE} : PKE challenge
c^{KEM} : KEM challenge
k : key
S : set

PKGEN()

assert $pk = \bot$
$pk \leftarrow\$ \mathsf{KEMGEN}()$
assert $sk = \bot$
$pk, sk \leftarrow\$ \Pi^{\mathsf{KEM}}.gen()$
return pk

PKENC(m)

assert $pk \neq \bot$
assert $c^{\mathsf{PKE}} = \bot$
$c_k \leftarrow\$ \mathsf{ENCAPS}()$
assert $pk \neq \bot$
assert $c^{\mathsf{KEM}} = \bot$
$c^{\mathsf{KEM}}, k' \leftarrow\$ \Pi^{\mathsf{KEM}}.encaps(pk)$
$\mathsf{SET}(k)$
assert $k = \bot$
$k \leftarrow k'$
$c_k \leftarrow c^{\mathsf{KEM}}$
$c_m \leftarrow\$ \mathsf{ENC}(m)$
$k \leftarrow \mathsf{GET}()$
assert $k \neq \bot$
if $b = 1$:
  $c_m \leftarrow\$ \Pi^{\mathsf{DEM}}.enc(k, 0^{|m|})$
else :
  $c_m \leftarrow\$ \Pi^{\mathsf{DEM}}.enc(k, m)$
$S \leftarrow S \cup \{c_m\}$
$c^{\mathsf{PKE}} \leftarrow (c_k, c_m)$
return c^{PKE}

PKDEC(c')

assert $pk \neq \bot$
assert $c^{\mathsf{PKE}} \neq c'$
$(c_k, c_m) \leftarrow$ **parse** c^{PKE}
$(c'_k, c'_m) \leftarrow$ **parse** c'
if $c'_k = c_k$:
  $m \leftarrow \mathsf{DEC}(c'_m)$
  assert $c'_m \notin S$
  $k \leftarrow \mathsf{GET}()$
  assert $k \neq \bot$
  $m \leftarrow \Pi^{\mathsf{DEM}}.dec(k, c'_m)$
else :
  $k' \leftarrow \mathsf{DECAPS}(c'_k)$
  assert $sk \neq \bot$
  assert $c^{\mathsf{KEM}} \neq c'_k$
  $k' \leftarrow \Pi^{\mathsf{KEM}}.decaps(sk, c'_k)$
  $m \leftarrow \Pi^{\mathsf{DEM}}.dec(k', c'_m)$
return m

$G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$

Parameters

b : idealization bit
Π^{KEM} : KEM scheme
Π^{DEM} : DEM scheme

State

pk : public key
sk : secret key
c : challenge

PKGEN()

assert $sk = \bot$
$pk, sk \leftarrow\$ \Pi^{\mathsf{PKE}}.gen()$
$pk, sk \leftarrow\$ \Pi^{\mathsf{KEM}}.gen()$
return pk

PKENC(m)

assert $pk \neq \bot$
assert $c = \bot$
if $b = 0$ **then**
  $c \leftarrow\$ \Pi^{\mathsf{PKE}}.enc(pk, m)$
  $c_k, k \leftarrow\$ \Pi^{\mathsf{KEM}}.encaps(pk)$
  $c_m \leftarrow\$ \Pi^{\mathsf{DEM}}.enc(k, m)$
  $c \leftarrow (c_k, c_m)$
else
  $c \leftarrow\$ \Pi^{\mathsf{PKE}}.enc(pk, 0^{|m|})$
  $c_k, k \leftarrow\$ \Pi^{\mathsf{KEM}}.encaps(pk)$
  $c_m \leftarrow\$ \Pi^{\mathsf{DEM}}.enc(k, 0^{|m|})$
  $c \leftarrow (c_{\mathsf{KEM}}, c_m)$
return c

PKDEC(c')

assert $sk \neq \bot$
assert $c \neq c'$
$m \leftarrow \Pi^{\mathsf{PKE}}.dec(sk, c')$
$(c'_k, c'_m) \leftarrow$ **parse** c'
$k \leftarrow \Pi^{\mathsf{KEM}}.decaps(sk, c'_k)$
$m \leftarrow \Pi^{\mathsf{DEM}}.dec(k, c'_m)$
return m

Figure 7: Games $\mathsf{Mod}^{b,0}$ and $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$

Fournet, Kohbrok, Kohlweiss (BDEFKK) are the first to introduce the concept and technique in their work [BDLE+21], they do not explicitly express, state, and prove this foundational observation but rather they implicitly use it in the proof of two major lemmata C.2 and C.5 of their work. These lemmata are later introduced in Section 3.3.3.

2.4 SMT Solvers and SMT-LIB language

Satisfiability Modulo Theory (SMT) problems are decision problems that ask for satisfiability of a set of given logical formulae. When the logical formulae come from propositional logic, these problems are called SAT problems. When the formulae become more complex and come from first (or higher) order logic, these problems are referred to as SMT problems. Their name comes from the fact that these problems usually ask for satisfiability of a set of formulae within a certain theory such as the real numbers or integers as well as data structures such as lists, arrays, strings, etc.

SMT problems are computationally very hard because SMT problems are generalization of SAT problems that are NP-complete. On the other hand, decision problem of natural numbers with multiplication and addition is undecidable. SMT solvers are tools that aim to solve SMT problems for a subset of inputs. CVC5 [BBB+22] and Z3 [DMB08] are, among others, two open source SMT solvers. In this thesis, we use SSBee which relies on CVC5. We also did a limited experiment with Z3 that we discuss in Section 2.6.6.

SMT-LIB [BFT16] is an initiative that, among others, standardizes a common input and output language SMT-LIB for SMT solvers. The SMT-LIB Standard proposes a syntax to define logical formulae and feed into SMT solvers. CVC5 and Z3 both have adopted the SMT-LIB language standard as their input language. We refer the reader to the latest standard version 2.7 [BFT16] for the syntax of the language. The file extension for the files written in SMT-LIB language is `.smt2`.

Given a set of formulae modulo some theory, SMT solvers have three possible outputs: `sat` indicating the satisfiability of the formulae followed by a model (assignments to all free variables of the formulae), `unsat` indicating unsatisfiability of the formulae and optionally (if requested by the user) a proof file, and `unknown` indicating inability of the solver to determine the satisfiability followed by a partial model that does not satisfy the formulae. The other possible outcome is that solver does not terminate. This is a very unfortunate case as the solver wastes the time of the user without giving any information even about its inability to determine the result. This case is usually controlled by setting a timeout for the solver.

As a final note, we want to elaborate on how predicates can be proven with SMT solvers. It is easy to see that a predicate p can be disproven by checking satisfiability of $\neg p$ and getting a counterexample (a model) from the solver. To prove p, one shall check unsatisfiability of $\neg p$. Looking forward, we usually expect to receive `unsat` from SSBee (i.e. the SMT solver backend) as we prove properties. However, if we receive a `sat`, it means the solver has found a concrete counterexample for our proofs. Having said that, we did not encounter `sat` results (counterexamples) very often in our verifications of TLS 1.3 or the KEM-DEM example, which made verification harder

with almost no help from the tool.

2.5 SSBee

SSBee is a formalization of SSP framework that automates SSP-style reduction proofs using SMT solvers with minimal help from the user. As illustrated in Section 2.1 with the proof of Theorem 2.4, SSP-style reduction proofs mainly consist of two kinds of game hops: code equivalence and computational equivalence. Computational equivalence of two big hybrid games is a standard reduction to indistinguishability of two small games (the security assumption) using Lemma 2.1 by presenting a reduction package $\mathcal{R}$ and proving two code equivalences $G^b_{big} \overset{code}{\equiv} \mathcal{R} \rightarrow G^b_{small}$ for $b \in \{0, 1\}$. For example, in the KEM-DEM paradigm example in the previous section, the small assumption games were the KEM-CCA and DEM-CCA security assumptions respectively for the KEM and DEM schemes. SSBee helps cryptographers to automate proof of their code equivalence game hops as well as verifying their reductions to security assumptions in computational equivalences.

SSBee defines a language close to the pseudocodes shown in Section 2.1 that allows the user to define their packages, security games (as compositions of these packages), and finally their security reductions as a sequence of game hops. SSBee verifies the computational equivalence game hops by checking that the call (composition) graph of the big security games can be split into a reduction package and the security assumption game. More importantly, SSBee verifies the code equivalence game hops by automating code-based arguments such as the one presented in 2.7 with an SMT solver. Relying on Theorem 2.3 and Corollary 2.3.1, SSBee requires the user to define a state relation and tries to prove the given state relation is invariant as well as the same-output property holds. To this end, SSBee translates these two proof obligations together with the code of packages into a set of first-order logic formulae in the SMT-LIB syntax and asks the SMT solver to prove the formulae are unsatisfiable.

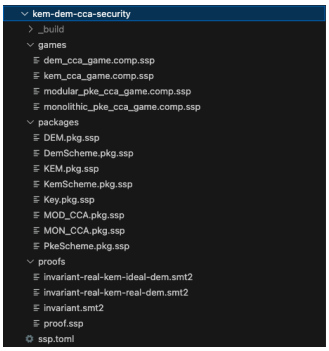


Figure 8: Theorem 2.4 formalization project directory in SSBee

We illustrate the language of SSBee together with its main concepts by formalizing the KEM-DEM security reduction in Theorem 2.4 in SSBee. Fortunately, we have laid down the theoretical foundation of SSBee in Section with the proof of Lemma 2.2 and Theorem 2.3. Theorem 2.4 is formalized in an SSBee project and its code is available online hosted on a GitHub Repository [Raj25a]. Figure 8 shows the KEM-DEM formalization SSBee project directory. An SSBee project is a directory including three subdirectories for the games, proofs, and packages as well as an empty file `ssp.toml` at the root of project. We begin by looking at how a security reduction as a sequence of game hops can be formalized in SSBee in a proof file.

2.5.1 Proofs in SSBee

The proof file `proof.ssp` (in `proofs` subdirectory) begins with the definition of proof parameters (or constants) including *abstract functions*. Listing below shows the beginning of the proof file:

```
proof Proof {
    const b: Bool;
    const len: fn Bits(*) -> Integer;
    const zeros: fn Integer -> Bits(*);
    const kem_gen: fn Bits(2000) -> (Bits(100), Bits(1000));
    const kem_encaps: fn Bits(3000), Bits(100) -> (Bits(256), Bits(400));
    const kem_decaps: fn Bits(1000), Bits(400) -> Bits(256);
    const dem_enc: fn Bits(500), Bits(256), Bits(*) -> Bits(*);
    const dem_dec: fn Bits(256), Bits(*) -> Bits(*);
```

Abstract functions are function declarations without body. If the user does not attach meaning to them through lemmata, state relations, or SMT assertions, they can be essentially considered as uninterpreted functions. One common use case for them is the declaration of a cryptographic scheme functions which we do not need any properties from them except for the scheme syntax and its correctness. For instance, key generation, encapsulation, and decapsulation functions of a KEM scheme or encryption and decryption functions of the DEM scheme can be modeled as abstract functions. We later on explain how KEM correctness can be expressed in SSBee. Abstract functions need a type for each of their arguments and their output. The type should be one of the builtin types in SSBee. At the time of writing this thesis, SSBee supports the following types:

Type	Syntax	Description	Literals
Integer	`Integer`	Non-negative integers	`0`, `256`, ...
Boolean	`Bool`	Boolean	`true`, `false`
Bitstring	`Bits(256)`	Bitstring of fixed length	No literals
Bitstring	`Bits(n)`	Bitstring of length `n`	No literals
Bitstring	`Bits(*)`	Bitstring of arbitrary length	No literals
Table	`Table(TInput, TOutput)`	Mapping from `TInput` to `TOutput`	No literals
Maybe	`Maybe(T)`	Option type	`None`, `Some(v)`
Tuple	`(T1, T2, ..., TN)`	Tuple of types `T1`, ..., `TN`	`(1, true)`

We will discuss the types Table, Maybe, and Tuple when illustrating the code of packages. SSBee currently does not support custom types or higher order functions (i.e. functions that receive or return a function). However, functions can receive and return Tables, Tuples, or Maybe types, important features that we will benefit from in Section 5 when encoding the Hilbert operator in SSBee-assisted proof of Lemma 5.20. Since SSBee does not support any literals for bitstrings, we use `zeros` function to generate all-zero strings of the given length. Moreover, SSBee does not consider any properties for bitstrings and they are translated to independent SMT-LIB data types. For example, SSBee generates custom SMT-LIB data types with names `Bits_*`, `Bits_128`, or `Bits_256` when compiling `Bits(*)`, `Bits(128)`, or `Bits(256)`. Therefore, any operations on and properties of bistrings, such as their concatenation or retrieving

their lengths, shall be handled explicitly by the user. We define function `len` to return the length of a bitstring of arbitrary length.

Recall that the functions $\Pi^{\mathsf{KEM}}.gen$, $\Pi^{\mathsf{KEM}}.encaps$, and $\Pi^{\mathsf{DEM}}.enc$ are randomized. Since abstract functions in SSBee are deterministic mathematical functions, we make the randomness explicit by adding an additional argument to these functions for the randomness string. Arguments with types `Bits(2000)`, `Bits(3000)`, and `Bits(500)` are randomness strings for $\Pi^{\mathsf{KEM}}.gen$, $\Pi^{\mathsf{KEM}}.encaps$, and $\Pi^{\mathsf{DEM}}.enc$, respectively. The choice of other bitstring lengths is not arbitrary. We have used different lengths for the public key (`Bits(100)`), secret key (`Bits(1000)`), encapsulated key (`Bits(400)`), encryption key (`Bits(256)`), and message (`Bits(*)`) in order to benefit from SSBee type checker and easily find key type mismatch bugs during compilation.

Moving forward, the rest of the proof file instantiates the SSP games and their parameters with concrete values (or possibly proof parameters) before stating the game hops. We will define two generic SSP games `ModularPkeCcaGame` and `MonolithicPkeCcaGame` corresponding to games Mod^{b_1,b_2} (defined in Section 2.5) and $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ (defined in Section 2.1.1), respectively. We refer to $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ as a monolithic game because it does not split its functionality into several modular packages. Listing below shows the instantiation of the real monolithic game $G^{0,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ and the first hybrid game $H_1 := \mathsf{Mod}^{0,0}$:

```
instance monolithic_pke_cca_real_game = MonolithicPkeCcaGame {
    params {
        b: false,
        len: len,
        zeros: zeros,
        kem_gen: kem_gen,
        kem_encaps: kem_encaps,
        kem_decaps: kem_decaps,
        dem_enc: dem_enc,
        dem_dec: dem_dec,
    }
}
instance modular_pke_cca_game_with_real_key_and_real_dem = ModularPkeCcaGame {
    params {
        key_idealization: false,
        dem_idealization: false,
        len: len,
        zeros: zeros,
        kem_gen: kem_gen,
        kem_encaps: kem_encaps,
        kem_decaps: kem_decaps,
        dem_enc: dem_enc,
        dem_dec: dem_dec,
    }
}
```

Notice that parameters `b`, `key_idealization`, and `dem_idealization` are instantiated with concrete values while other game parameters are function parameters and are instantiated with the abstract functions defined in the beginning of the proof file as proof parameters.

The ideal monolithic game $G^{1,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ as well as hybrid games $H_2 := \mathtt{Mod}^{0,1}, H_3 := \mathtt{Mod}^{1,1}, H_4 := \mathtt{Mod}^{1,0}$ are instantiated similarly.

The last step before stating the game hops is to state the KEM-CCA and DEM-CCA security games as well as our security assumption of their indistinguishability.

```
instance kem_cca_game_real = KemCcaGame {
    params {
        b: false,
        kem_gen: kem_gen,
        kem_encaps: kem_encaps,
        kem_decaps: kem_decaps,
    }
}
instance kem_cca_game_ideal = KemCcaGame {
    params {
        b: true,
        kem_gen: kem_gen,
        kem_encaps: kem_encaps,
        kem_decaps: kem_decaps,
    }
}
```

```
instance dem_cca_game_real = DemCcaGame {
    params {
        b: false,
        len: len,
        zeros: zeros,
        dem_enc: dem_enc,
        dem_dec: dem_dec,
    }
}
instance dem_cca_game_ideal = DemCcaGame {
    params {
        b: true,
        len: len,
        zeros: zeros,
        dem_enc: dem_enc,
        dem_dec: dem_dec,
    }
}
```

We then state the assumptions as follows:

```
    assumptions {
        KEM_CCA_Security: kem_cca_game_real ~ kem_cca_game_ideal
        DEM_CCA_Security: dem_cca_game_real ~ dem_cca_game_ideal
    }
```

Finally, we state the game hops. Recall from Section 2.5 that we prove the equivalence of real and ideal monolithic games $G^{0,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ and $G^{1,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ via 5 game hops: the first and last ones are code equivalence while the three middle hops are computational equivalence by a standard reduction to KEM-CCA or DEM-CCA security assumptions.

$$G^{0,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}} \overset{code}{\equiv} H_1 \overset{comp}{\approx} H_2 \overset{comp}{\approx} H_3 \overset{comp}{\approx} H_4 \overset{code}{\equiv} G^{1,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$$

This is stated in SSBee as follows:

```
gamehops {
    equivalence monolithic_pke_cca_real_game
    modular_pke_cca_game_with_real_kem_and_real_dem {
        ...
    }

    reduction modular_pke_cca_game_with_real_kem_and_real_dem
    modular_pke_cca_game_with_ideal_key_and_real_dem {
        assumption KEM_CCA_Security
        ...
    }

```

```
    reduction modular_pke_cca_game_with_ideal_key_and_real_dem
    modular_pke_cca_game_with_ideal_key_and_ideal_dem {
        assumption DEM_CCA_Security
        ...
    }

    reduction modular_pke_cca_game_with_ideal_key_and_ideal_dem
    modular_pke_cca_game_with_real_kem_and_ideal_dem {
        assumption KEM_CCA_Security
        ...
    }

    equivalence monolithic_pke_cca_ideal_game
    modular_pke_cca_game_with_real_kem_and_ideal_dem {
        ...
    }
}
```

Listing 1: Proof file

Before diving into the game hops, we describe how our SSP games and packages can be written in SSBee.

2.5.2 Games and packages in SSBee

We begin by illustrating how SSP packages can be formalized in SSBee. Listing below shows the SSBee code of package $\mathsf{KEY}^{b,\lambda=256}$.

```
package Key {
    params {
        b: Bool
    }

    state {
        k: Maybe(Bits(256))
    }

    oracle SET(kp: Bits(256)) {
        assert (k == None);
        if b {
            k1 <-$ Bits(256);
            k <- Some(k1);
        } else {
            k <- Some(kp);
        }
    }

    oracle GET() -> Bits(256) {
        assert (k != None as Bits(256));
        return Unwrap(k);
    }
}
```

A package definition is similar to a class in a programming language. The `state` code block describes all the private fields (variables, tables, etc.) of the package. For

example, `k` stores the key with type `Maybe(Bits(256))`. *Maybe* type is used to indicate the key may be null. SSBee does not make any assumption on the initial values of the fields. Specifically one should assume a *Maybe* type may contain a null or some value. Looking ahead to the code equivalence proofs in SSBee and reminding the fundamental theorem of code equivalence 2.3, when proving code equivalence obligations such as same-output property or invariance of state relations, the only assumption about the states of the left and right games is the state relations. However, it is also crucial to prove the state relation I holds for the initial states of the games (S_0^l, S_0^r), i.e. $I(S_0^l, S_0^r)$. This is currently a missing feature of SSBee and one has to manually define the initial states of the games and write down the proof obligation to verify $I(S_0^l, S_0^r)$ directly using the SMT-LIB language.

Apart from the state, a package can declare some parameters, such as idealization bits. As we will see in the next section, games have to instantiate packages with concrete parameters. Analogous to class analogy for the packages, instances of a package correspond to objects of a class. Parameters can be used as constants or readonly variables, which can not be assigned to, in the code of oracles of the package, but they are not global constants or and can vary between instances of a package. An integer parameter `n` can be a special type parameter if it is used as the argument of a `Bits(n)` type in the code of oracle. In such a situation, SSBee generates different codes for different concrete values of n with which the package is instantiated.

Finally, a package defines all the oracles of its output interface. Oracles may not return a value such as the oracle `SET` shown above. However, oracles with a return should always return a value. SSBee uses the familiar notation `<-$` for random samplings and `<-` for regular assignments. Currently, SSBee only supports sampling bitstrings. `Some(...)` and `None` are two constructors for the *Maybe* type. The operation `Unwrap` retrieves the value stored in a *Maybe* type if it is not none. It is crucial to take into account that c can abort the oracle if the value is none. To achieve this semantics, SSBee inserts well-definedness assertions before `Unwrap` in the compiled SMT-LIB code. Looking ahead again to the code equivalence proofs, it is a common mistake when verifying proofs in SSBee to overlook these hidden assertions. As a result, SSBee may fail to prove the same-output property because one oracle, say the left one, aborts due to an `Unwrap` operation while the right oracle returns a value. The best solution that brings visibility to the proof is to precede the `Unwrap` operations with if conditions or plain assertions to check for possible null value of the type.

With this introduction we demonstrate other packages useful in the KEM-DEM formalization project. The following listing show the code of stateless package `KemScheme`. We mentioned earlier that we make the randomness used by the KEM scheme functions $\Pi^{\mathsf{KEM}}.gen$, $\Pi^{\mathsf{KEM}}.encaps$, and $\Pi^{\mathsf{DEM}}.enc$ explicit. Therefore, in order to use these functions, one has to first sample a random string of proper length (2000 bits for $\Pi^{\mathsf{KEM}}.gen$ and 3000 bits for $\Pi^{\mathsf{KEM}}.encaps$) and then feed into to the functions along possibly their other arguments (e.g. the public key pk for $\Pi^{\mathsf{KEM}}.encaps$). One reason for making randomness explicit is that SSBee only supports randomness sampling for bitstrings while abstract functions are deterministic. Stateless package `KemScheme` hides the two-step process of sampling and key computations from the scheme users. In general, stateless packages are a common technique to reuse blocks of repeatable

around the codebase of an SSBee project.

```
package KemScheme {
    params {
        kem_gen: fn Bits(2000) -> (Bits(100), Bits(1000)),
        kem_encaps: fn Bits(3000), Bits(100) -> (Bits(256), Bits(400)),
        kem_decaps: fn Bits(1000), Bits(400) -> Bits(256),
    }

    oracle KEM_GEN() -> (Bits(100), Bits(1000)) {
        r <-$ Bits(2000);
        return kem_gen(r);
    }

    oracle KEM_ENCAPS(pk: Bits(100)) -> (Bits(256), Bits(400), Bits(3000)) {
        r <-$ Bits(3000);
        k_ek <- kem_encaps(r, pk);
        (k, ek) <- parse k_ek;
        return (k, ek, r);
    }

    oracle KEM_DECAPS(sk: Bits(1000), ek: Bits(400)) -> Bits(256) {
        return kem_decaps(sk, ek);
    }
}
```

The package requires the KEM scheme abstract functions and declares them as parameters. These parameters are instantiated using the proof constants mentioned earlier. Observe that the oracle KEM_ENCAPS returns the sampled random string to the scheme user. Although this may sound strange, we will not expose this information to the adversary, but rather use it only for verification purposes as well as simplifying state relations in the code equivalence proofs. The first usecase of the randomness string appears in the next package we introduce: KEM.

```
package KEM {
    state {
        pk: Maybe(Bits(100)),
        sk: Maybe(Bits(1000)),
        ek: Maybe(Bits(400)),
        encaps_randomness: Maybe(Bits(3000))
    }

    import oracles {
        SET(k: Bits(256)),
        KEM_GEN() -> (Bits(100), Bits(1000)),
        KEM_ENCAPS(pk: Bits(100)) -> (Bits(256), Bits(400), Bits(3000)),
        KEM_DECAPS(sk: Bits(1000), ek: Bits(400)) -> Bits(256),
    }

    oracle KEMGEN() -> Bits(100) {
        assert (sk == None);
        pk_sk <- invoke KEM_GEN();
        (pk1, sk1) <- parse pk_sk;
        pk <- Some(pk1);
```

```
21          sk <- Some(sk1);
22          return pk1;
23      }
24
25      oracle ENCAPS() -> Bits(400) {
26          assert (pk != None as Bits(100));
27          assert (ek == None);
28          k_ek_r <- invoke KEM_ENCAPS(Unwrap(pk));
29          (k, ek1, r) <- parse k_ek_r;
30          encaps_randomness <- Some(r);
31          ek <- Some(ek1);
32          _ <- invoke SET(k);
33          return ek1;
34      }
35
36      oracle DECAPS(encapsk: Bits(400)) -> Bits(256) {
37          assert (sk != None as Bits(1000));
38          assert (encapsk != Unwrap(ek));
39          k <- invoke KEM_DECAPS(Unwrap(sk), encapsk);
40          return k;
41      }
42  }
```

Compared to the pseudocode of KEM, SSBee code of KEM defines variable ek for the challenge and an additional state variable encaps_randomness, which stores the internal random string r sampled by the KEM_ENCAPS oracle. See line 28-30 where the state variables are assigned to. Although encaps_randomness is not necessary for the main proof, but it is very useful for the verification and makes the state relations simpler. Notice that encaps_randomness is also not stored in the package KemScheme because we want that package to be stateless and expose ready-to-use general scheme function. Moreover, ENCAPS oracle samples an encapsulated key only once. Further queries to ENCAPS will abort at line 27. Therefore, storing the random string encaps_randomness is sound. We borrowed the idea of such state variables from program verification, where they are called *ghost* variables. It is important to make sure that ghost variables should only be used for verification and blindly removing all the lines containing any ghost variable should not change the semantics of the program (or oracle).

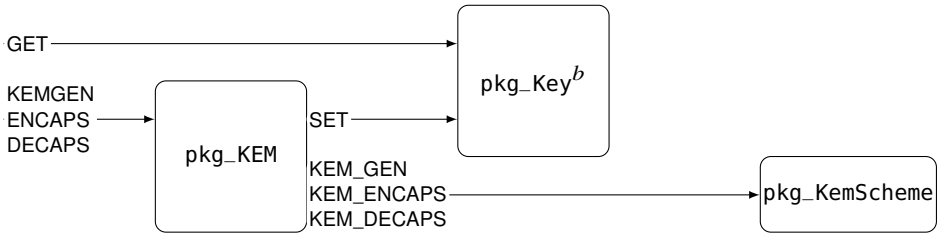
Observe that the package imports four oracles as its input interface. The SET oracle comes from the package Key while the other three come from the package KemScheme. Imported oracles are invoked with the keyword invoke. The parse operation is used to destructure tuples and access their elements.

With these three package, we can now present the definition of the KEM-CCA security game $G^{b,\Pi^{\lambda=256,\mathsf{KEM}}}_{\mathsf{KEM\text{-}CCA}}$ in SSBee. Listing below shows the definition of composition KemCcaGame.

```
composition KemCcaGame {
    const b: Bool;
    const kem_gen: fn Bits(2000) -> (Bits(100), Bits(1000));
    const kem_encaps: fn Bits(3000), Bits(100) -> (Bits(256), Bits(400));
    const kem_decaps: fn Bits(1000), Bits(400) -> Bits(256);

    instance pkg_KemScheme = KemScheme {
```

```
        params {
            kem_gen: kem_gen,
            kem_encaps: kem_encaps,
            kem_decaps: kem_decaps,
        }
    }

    instance pkg_Key = Key {
        params {
            b: b
        }
    }

    instance pkg_KEM = KEM {
    }

    compose {
        pkg_KEM: {
            SET: pkg_Key,
            KEM_GEN: pkg_KemScheme,
            KEM_ENCAPS: pkg_KemScheme,
            KEM_DECAPS: pkg_KemScheme
        }
        adversary: {
            KEMGEN: pkg_KEM,
            ENCAPS: pkg_KEM,
            DECAPS: pkg_KEM,
            GET: pkg_Key
        }
    }
}
```

A game definition begins with declaration of game parameters (idealization bits, functions, etc) and is followed by the instantiation of packages with concrete parameters. Notice that the package `Key` is instantiated with the game parameter `b`. Finally, the call graph of the packages are communicated to SSBee with a composition mapping. Users has to determine which package provides each of the oracles in the input interface of packages. The keyword `adversary` helps to determine which oracles are exposed to the adversary. Games can also be viewed as templates which are then instantiated with concrete proof constants in the proof file. Using the package and game definitions, SSBee exports the following call graph together with its TikZ code: (package parameter is added manually)



Similar to the package `KemScheme`, we introduce the stateless package `DemScheme` which

hides the sampling process for the DEM scheme encryption function. Random strings with 500 bits are sampled and fed into the abstract function `dem_enc`.

```
package DemScheme {
    params {
        dem_enc: fn Bits(500), Bits(256), Bits(*) -> Bits(*),
        dem_dec: fn Bits(256), Bits(*) -> Bits(*)
    }

    oracle DEM_ENC(k: Bits(256), m: Bits(*)) -> Bits(*) {
        r <-$ Bits(500);
        return dem_enc(r, k, m);
    }

    oracle DEM_DEC(k: Bits(256), c: Bits(*)) -> Bits(*) {
        return dem_dec(k, c);
    }
}
```

Notice the type `Bits(*)` used for messages and ciphertexts of arbitrary length. Next, we present the SSBee code of the package $\texttt{DEM}^{b,\lambda=256}$.

```
package DEM {
    params {
        b: Bool,
        len: fn Bits(*) -> Integer,
        zeros: fn Integer -> Bits(*),
    }

    state {
        T: Table(Bits(*), Bool)
    }

    import oracles {
        GET() -> Bits(256),
        DEM_ENC(k: Bits(256), m: Bits(*)) -> Bits(*),
        DEM_DEC(k: Bits(256), c: Bits(*)) -> Bits(*),
    }

    oracle ENC(m: Bits(*)) -> Bits(*) {
        k <- invoke GET();
        if b {
            c <- invoke DEM_ENC(k, zeros(len(m)));

        } else {
            c <- invoke DEM_ENC(k, m);
        }
        T[c] <- Some(true);
        return c;
    }

    oracle DEC(c: Bits(*)) -> Bits(*) {
        if (T[c] != None as Bool) {
            abort;
        }
```

```
        k <- invoke GET();
        m <- invoke DEM_DEC(k, c);
        return m;
    }
}
```

The package defines a table `T` (instead of set S in $\texttt{DEM}^{b,\lambda=256}$) to store the generated challenges. Table `T` is a mapping from bitstrings of arbitrary length to boolean values. SSBee uses the type `Maybe(Bool)` for the entries of the table, allowing to model empty entries of the table. That is why the table entries are assigned to with `Some(...)` constructor. In our case, out of three possible values for the entries of the table (null, true, and false), we only have null or true entries, simulating the set inclusion indicator function.

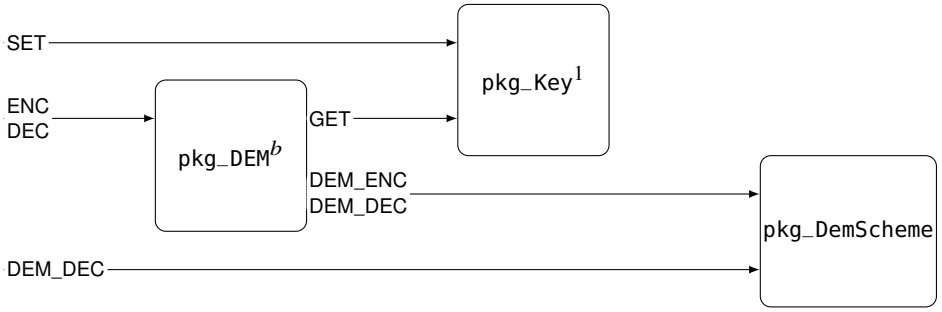
The packages uses the abstract function `zeros` to generate an all-zero bitstring of the given length because SSBee does not support bitstring literals at the time of writing this thesis.

With these packages, we can define the DEM-CCA security game $G^{b,\Pi^{\lambda=256,\texttt{DEM}}}_{\texttt{DEM-CCA}}$ in SSBee. Listing below shows the definition of composition `DemCcaGame`.

```
composition DemCcaGame {
    const b: Bool;

    const len: fn Bits(*) -> Integer;
    const zeros: fn Integer -> Bits(*);
    const dem_enc: fn Bits(500), Bits(256), Bits(*) -> Bits(*);
    const dem_dec: fn Bits(256), Bits(*) -> Bits(*);

    instance pkg_DemScheme = DemScheme {
        params {
            dem_enc: dem_enc,
            dem_dec: dem_dec
        }
    }

    instance pkg_Key = Key {
        params {
            b: true
        }
    }

    instance pkg_DEM = DEM {
        params {
            b: b,
            len: len,
            zeros: zeros,
        }
    }

    compose {
        pkg_DEM: {
            DEM_ENC: pkg_DemScheme,
            DEM_DEC: pkg_DemScheme,
            GET: pkg_Key
```

```
        }
        adversary: {
            ENC: pkg_DEM,
            DEC: pkg_DEM,
            SET: pkg_Key,
            DEM_DEC: pkg_DemScheme
        }
    }
}
```

The structure of the game is very similar to `KemCcaGame`. Notice that the adversary is given access to the oracle `DEM_DEC` of the stateless package `DemScheme`. This is safe because the adversary normally has access to and knows the internals of the KEM, DEM, and PKE schemes. Moreover, the package `Key` is idealized when instantiated with b = **true**. SSBee automatically generates the following call graph for this game.



Similar to KEM and DEM schemes defines ad stateless packages `KemScheme` and `DemScheme`, we define the following stateless package for a hybrid PKE construction from a KEM and a DEM scheme.

```
package PkeScheme {
    import oracles {
        KEM_GEN() -> (Bits(100), Bits(1000)),
        KEM_ENCAPS(pk: Bits(100)) -> (Bits(256), Bits(400), Bits(3000)),
        KEM_DECAPS(sk: Bits(1000), ek: Bits(400)) -> Bits(256),
        DEM_ENC(k: Bits(256), m: Bits(*)) -> Bits(*) /* ciphertext length */,
        DEM_DEC(k: Bits(256), c: Bits(*) /* ciphertext length */) -> Bits(*),
    }

    oracle GEN() -> (Bits(100), Bits(1000)) {
        pk_sk <- invoke KEM_GEN();
        return pk_sk;
    }

    oracle ENC(pk: Bits(100), m: Bits(*)) -> (Bits(400), Bits(*), Bits(3000)) {
        k_ek_r <- invoke KEM_ENCAPS(pk);
        (k, ek, r) <- parse k_ek_r;
        c <- invoke DEM_ENC(k, m);
        return (ek, c, r);
    }

    oracle DEC(sk: Bits(1000), c: (Bits(400), Bits(*))) -> Bits(*) {
        (ek, ctxt) <- parse c;
```

```
        k <- invoke KEM_DECAPS(sk, ek);
        m <- invoke DEM_DEC(k, ctxt);
        return m;
    }
}
```

Observe the package imports oracles for the KEM and DEM scheme functionalities.

Since we wish to prove the code equivalence of $\texttt{Mod}^{b,0}$ and $G^{b,\Pi^{\text{PKE}}}_{\text{PKE-CCA}}$ for $b \in \{0,1\}$ in SSBee, we need to define the modular game $\texttt{Mod}^{b_1,b_2}$ and monolithic game $G^{b,\Pi^{\text{PKE}}}_{\text{PKE-CCA}}$. In order to define the modular game $\texttt{Mod}^{b_1,b_2}$, we need to first define the package `Comb` in SSBee. We call this package `MOD_CCA` in our SSBee project because it is a modular combiner of packages `KEM` and `DEM`.

```
package MOD_CCA {
    import oracles {
        KEMGEN() -> Bits(100),
        ENCAPS() -> Bits(400),
        DECAPS(ek: Bits(400)) -> Bits(256),
        ENC(m: Bits(*)) -> Bits(*),
        DEC(c: Bits(*)) -> Bits(*),
        DEM_DEC(k: Bits(256), c: Bits(*) /* ciphertext length */) -> Bits(*),
    }

    state {
        pk: Maybe(Bits(100)),
        c: Maybe((Bits(400),Bits(*))),
        ek: Maybe(Bits(400)),
        em: Maybe(Bits(*))
    }

    oracle PKGEN() -> Bits(100) {
        assert (pk == None);
        pk1 <- invoke KEMGEN();
        pk <- Some(pk1);
        return pk1;
    }

    oracle PKENC(m: Bits(*)) -> (Bits(400),Bits(*)) {
        assert (pk != None as Bits(100));
        assert (c == None);
        ek1 <- invoke ENCAPS();
        ek <- Some(ek1);
        em1 <- invoke ENC(m);
        em <- Some(em1);
        c1 <- (ek1, em1);
        c <- Some(c1);
        return c1;
    }

    oracle PKDEC(ek_ctxt: (Bits(400),Bits(*))) -> Bits(*) {
        assert (pk != None as Bits(100));
        assert (Unwrap(c) != ek_ctxt);
        (encapsk, ctxt) <- parse ek_ctxt;
```

```
41          if (encapsk == Unwrap(ek)) {
42              m <- invoke DEC(ctxt);
43          } else {
44              k <- invoke DECAPS(encapsk);
45              m <- invoke DEM_DEC(k, ctxt);
46          }
47          return m;
48      }
49  }
```

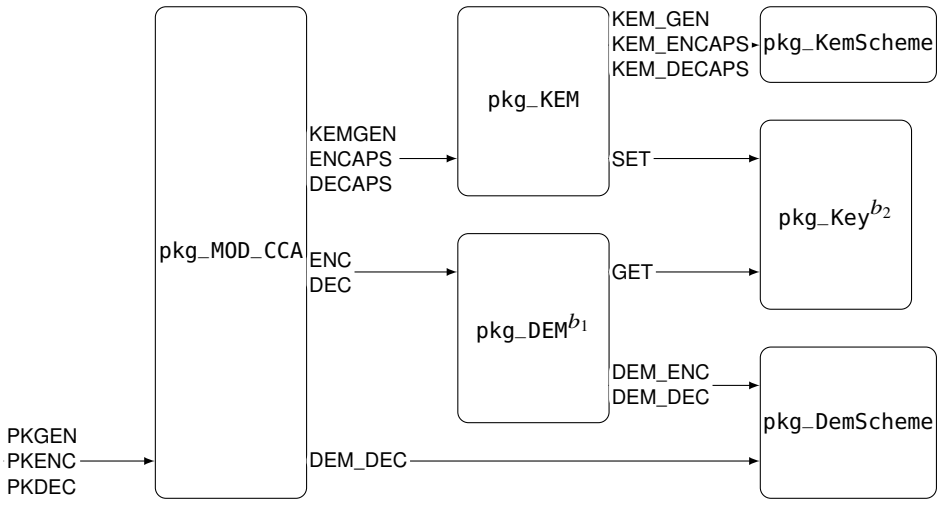
For the ease of expressing state relations, MOD_CCA stores the encapsulated key ek returned by ENCAPS and the encrypted message em returned by ENC separately, although they are both included in the challenge cipher text c. They can be considered ghost variable because they are used for ease of expressing the state relations. However, the oracle PKDEC should be modified not to rely on them. Notice how the package imports the oracle DEM_DEC to call at line 45 when the adversary queries MOD_CCA to decrypt a ciphertext with a different encapsulated key than the challenge ciphertext.

We can now define the modular game Mod^{b_1,b_2} as follows.

```
composition ModularPkeCcaGame {
    const key_idealization: Bool;
    const dem_idealization: Bool;
    const len: fn Bits(*) -> Integer;
    const zeros: fn Integer -> Bits(*);
    const kem_gen: fn Bits(2000) -> (Bits(100), Bits(1000));
    const kem_encaps: fn Bits(3000), Bits(100) -> (Bits(256), Bits(400));
    const kem_decaps: fn Bits(1000), Bits(400) -> Bits(256);
    const dem_enc: fn Bits(500), Bits(256), Bits(*) -> Bits(*);
    const dem_dec: fn Bits(256), Bits(*) -> Bits(*);
    instance pkg_Key = Key {
        params {
            b: key_idealization,
        }
    }
    instance pkg_DemScheme = DemScheme {
        params {
            dem_enc: dem_enc,
            dem_dec: dem_dec
        }
    }
    instance pkg_KemScheme = KemScheme {
        params {
            kem_gen: kem_gen,
            kem_encaps: kem_encaps,
            kem_decaps: kem_decaps,
        }
    }
    instance pkg_DEM = DEM {
        params {
            b: dem_idealization,
            len: len,
            zeros: zeros,
        }
```

```
    }
    instance pkg_KEM = KEM {}
    instance pkg_MOD_CCA = MOD_CCA {}
    compose {
        pkg_KEM: {
            KEM_GEN: pkg_KemScheme,
            KEM_ENCAPS: pkg_KemScheme,
            KEM_DECAPS: pkg_KemScheme,
            SET: pkg_Key
        }
        pkg_DEM: {
            DEM_ENC: pkg_DemScheme,
            DEM_DEC: pkg_DemScheme,
            GET: pkg_Key
        }
        pkg_MOD_CCA: {
            KEMGEN: pkg_KEM,
            ENCAPS: pkg_KEM,
            DECAPS: pkg_KEM,
            ENC: pkg_DEM,
            DEC: pkg_DEM,
            DEM_DEC: pkg_DemScheme
        }
        adversary: {
            PKGEN: pkg_MOD_CCA,
            PKENC: pkg_MOD_CCA,
            PKDEC: pkg_MOD_CCA
        }
    }
}
```

SSBee visualizes this composition as follows:



In order to define the monolithic security game $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$, we first introduce the monolithic combiner package `MON_CCA` that directly utilizes the oracles of `PkeScheme`. Recall that we defined the game $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ as a single package parameterized by the

PKE scheme. Package `MON_CCA` uses the same code but instead of being parameterized with PKE abstract function, imports the oracles of `PkeScheme`.

```
package MON_CCA {
    params {
        b: Bool,
        len: fn Bits(*) -> Integer,
        zeros: fn Integer -> Bits(*),
    }

    import oracles {
        GEN() -> (Bits(100), Bits(1000)),
        ENC(pk: Bits(100), m: Bits(*)) -> (Bits(400),Bits(*),Bits(3000)) /* ciphertext
     type */,
        DEC(sk: Bits(1000), c: (Bits(400),Bits(*)) /* ciphertext type */) -> Bits(*),
    }

    state {
        pk: Maybe(Bits(100)),
        sk: Maybe(Bits(1000)),
        c: Maybe((Bits(400),Bits(*)))
    }

    oracle PKGEN() -> Bits(100) {
        assert (sk == None);
        pk_sk <- invoke GEN();
        (pk1, sk1) <- parse pk_sk;
        pk <- Some(pk1);
        sk <- Some(sk1);
        return pk1;
    }

    oracle PKENC(m: Bits(*)) -> (Bits(400),Bits(*)) {
        assert (pk != None as Bits(100));
        assert (c == None);
        if b {
            c1 <- invoke ENC(Unwrap(pk), zeros(len(m)));
        } else {
            c1 <- invoke ENC(Unwrap(pk), m);
        }
        (ek, c2, r) <- parse c1;
        c <- Some((ek, c2));
        return (ek, c2);
    }

    oracle PKDEC(ek_ctxt: (Bits(400),Bits(*))) -> Bits(*) {
        assert (sk != None as Bits(1000));
        assert (Unwrap(c) != ek_ctxt);
        m <- invoke DEC(Unwrap(sk), ek_ctxt);
        return m;
    }
}
```

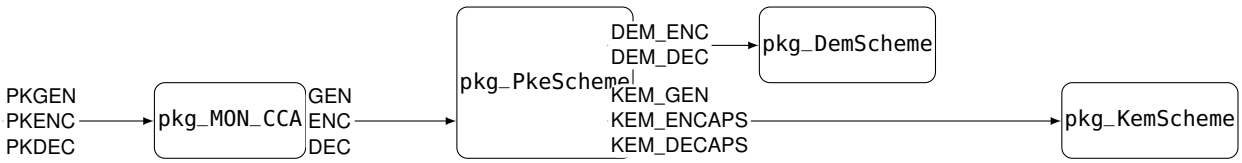
Notice that the package itself is parameterized with an idealization bit b analogous

to definition of $G^{b,\Pi^{\text{PKE}}}_{\text{PKE-CCA}}$ as a standalone monolithic parametrized package. Finally, we define the security game $G^{b,\Pi^{\text{PKE}}}_{\text{PKE-CCA}}$ as follows:

```
composition MonolithicPkeCcaGame {
    const b: Bool;
    const len: fn Bits(*) -> Integer;
    const zeros: fn Integer -> Bits(*);
    const kem_gen: fn Bits(2000) -> (Bits(100), Bits(1000));
    const kem_encaps: fn Bits(3000), Bits(100) -> (Bits(256), Bits(400));
    const kem_decaps: fn Bits(1000), Bits(400) -> Bits(256);
    const dem_enc: fn Bits(500), Bits(256), Bits(*) -> Bits(*);
    const dem_dec: fn Bits(256), Bits(*) -> Bits(*);

    instance pkg_KemScheme = KemScheme {
        params {
            kem_gen: kem_gen,
            kem_encaps: kem_encaps,
            kem_decaps: kem_decaps,
        }
    }

    instance pkg_DemScheme = DemScheme {
        params {
            dem_enc: dem_enc,
            dem_dec: dem_dec
        }
    }

    instance pkg_PkeScheme = PkeScheme {
    }

    instance pkg_MON_CCA = MON_CCA {
        params {
            b: b,
            len: len,
            zeros: zeros,
        }
    }

    compose {
        pkg_PkeScheme: {
            KEM_GEN: pkg_KemScheme,
            KEM_ENCAPS: pkg_KemScheme,
            KEM_DECAPS: pkg_KemScheme,
            DEM_ENC: pkg_DemScheme,
            DEM_DEC: pkg_DemScheme
        }
        pkg_MON_CCA: {
            GEN: pkg_PkeScheme,
            ENC: pkg_PkeScheme,
            DEC: pkg_PkeScheme
        }
        adversary: {
            PKGEN: pkg_MON_CCA,
```

```
            PKENC: pkg_MON_CCA,
            PKDEC: pkg_MON_CCA
        }
    }
}
```

SSBee visualizes the game as follows:



2.6 Proofs in SSBee: Revisited

Looking back to the game hops, there were five game hops from $G^{0,\Pi^{\text{PKE}}}_{\text{PKE-CCA}}$ to $G^{1,\Pi^{\text{PKE}}}_{\text{PKE-CCA}}$. The first and last game hops were code equivalence and the middle three were computational game hops (reductions to KEM-CCA and DEM-CCA security).

2.6.1 Reductions

We illustrate one of the computational game hops (reductions) as others are structured analogously. Consider the first computational game hop between the first two hybrid games $H_1 := \texttt{Mod}^{0,0}$ and $H_2 := \texttt{Mod}^{0,1}$. Indistinguishability of H_1 and H_2 reduces to the KEM-CCA security (i.e. indistinguishability of the games $G^{b,\Pi^{\text{KEM}}}_{\text{KEM-CCA}}$ for $b \in \{0,1\}$). As proved in the Claim 2.5, $H_{1+b} \overset{code}{\equiv} \mathcal{R}_1 \rightarrow G^{b,\Pi^{\text{KEM}}}_{\text{KEM-CCA}}$ for $b \in \{0,1\}$. Reduction $\mathcal{R}_1$ (visualized in Figure 6) are cuts in the call graphs of the games $\texttt{Mod}^{0,0}$ and $\texttt{Mod}^{0,1}$ such that when removed, games $G^{b,\Pi^{\text{KEM}}}_{\text{KEM-CCA}}$ are left.

To formally verify the reduction from the games H_{1+b} to $G^{b,\Pi^{\text{KEM}}}_{\text{KEM-CCA}}$ with the reduction package $\mathcal{R}_1$ in SSBee, we need to give a hint to SSBee how the reduction package $\mathcal{R}_1$ is constructed. We give this information by describing how the packages of H_{1+b} are mapped to the corresponding packages packages of the assumption games $G^{b,\Pi^{\text{KEM}}}_{\text{KEM-CCA}}$ when $\mathcal{R}_1$ is removed. As a result, all other packages can be composed and considered to be the reduction $\mathcal{R}_1$.

```
/* First hop idealizes KEM and reduces to KEM-CCA security */
reduction modular_pke_cca_game_with_real_kem_and_real_dem
    modular_pke_cca_game_with_ideal_kem_and_real_dem {
    assumption KEM_CCA_Security

    map kem_cca_game_real modular_pke_cca_game_with_real_kem_and_real_dem {
        pkg_KemScheme: pkg_KemScheme
        pkg_Key: pkg_Key
        pkg_KEM: pkg_KEM
    }

    map kem_cca_game_ideal modular_pke_cca_game_with_ideal_kem_and_real_dem {
```

```
        pkg_KemScheme: pkg_KemScheme
        pkg_Key: pkg_Key
        pkg_KEM: pkg_KEM
    }
}
```

For each mapping, SSBee expects one of the real or ideal games of the assumption is mapped to the one of the given games. Moreover, each mapping shall be bijective: it maps each package instances of the assumption game to a distinct package instance of the given game. Notice that it is only due to our naming covention that mapped packages have the same name. Generally speaking, the structure of a reduction from a game pair (G_0, G_1) to a assumption game pair (A_0, A_1) is as follows:

```
assumptions {
    A: A0 ~ A1
    ...
}
...
reduction G0 G1 {
    assumption A

    map A0 G0 {
        PA0_1: PG0_1,
        PA0_2: PG0_2,
        ...
    }
    map A1 G1 {
        PA1_1: PG1_1,
        PA1_2: PG1_2,
        ...
    }
}
```

For each pair of mapped games (e.g. $A_0 \mapsto G_0$), SSBee verifies that mapped package instances (e.g. `PA0_1` and `PG0_1`) are (1) instances of the same package and (2) instantiated with the same parameters. As a result, mapped packages have the same output interface. SSBee does not need to generate SMT-LIB code to verify these properties and checks these conditions algorithmically.

2.6.2 Code equivalence

We now look into how a code equivalence game hop can be verified in SSBee. Instead of proving each of the game hops $G^{0,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}} \overset{code}{\equiv} H_1$ or $G^{1,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}} \overset{code}{\equiv} H_4$ separately as shown in the Listing 1 previously, we prove both at the same time using a proof constant b. (We used the same technique in proof of Claim 2.7 and proved $G^l := \mathtt{Mod}^{b,0}$ is code equivalent to $G^r := G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$.) We instantiate the idealization parameters of the monolithic game $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ and modular game $\mathtt{Mod}^{b,0}$ with the proof constant b instead of concrete values **true** or **false**.

```
proof Proof {
    const b: Bool;
```

```
    instance monolithic_pke_cca_game = MonolithicPkeCcaGame {
        params {
            b: b,
            len: len,
            zeros: zeros,
            kem_gen: kem_gen,
            kem_encaps: kem_encaps,
            kem_decaps: kem_decaps,
            dem_enc: dem_enc,
            dem_dec: dem_dec,
        }
    }
    instance modular_pke_cca_game_with_real_kem = ModularPkeCcaGame {
        params {
            key_idealization: false,
            dem_idealization: b,
            len: len,
            zeros: zeros,
            kem_gen: kem_gen,
            kem_encaps: kem_encaps,
            kem_decaps: kem_decaps,
            dem_enc: dem_enc,
            dem_dec: dem_dec,
        }
    }
    ...
}
```

Game `monolithic_pke_cca_game` corresponds to $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ while `modular_pke_cca_game_with_real_kem` corresponds to $\mathtt{Mod}^{b,0}$. Notice that the same constant b is used to instantiate both games. We then prove the equivalence of the games as follows:

```
equivalence monolithic_pke_cca_game modular_pke_cca_game_with_real_kem {
    PKGEN: {
        invariant: [
            ./proofs/invariant.smt2
        ]

        lemmas {
            invariant: [no-abort]
            same-output: [no-abort]
            equal-aborts: []
        }
    }

    PKENC: {
        invariant: [
            ./proofs/invariant.smt2
        ]

        lemmas {
            invariant: [no-abort, lemma-rand-is-eq]
            same-output: [no-abort, lemma-rand-is-eq]
            equal-aborts: []
```

```
        }
    }

    PKDEC: {
        invariant: [
            ./proofs/invariant.smt2
        ]

        lemmas {
            invariant: [no-abort]
            same-output: [no-abort, lemma-kem-correctness]
            equal-aborts: []
        }
    }
}
```

Listing 2: Code equivalence in SSBee

SSBee relies on the Fundamental Theorem of Code Equivalence (Theorem 2.3) to prove the code equivalence of the given games. Namely, it asks the user to proivde a state relation I and a randomness mapping M using the SMT-LIB language and then tries to prove the same-output property for each oracle exposed by the games ad well as the invariance of the state relation I. Additionally, it allows the user to express lemmata to prove each of the proof obligations. Clearly, the proof is only sound when all the lemmata themselves are also proved.

For each of the oracles exposed by the games $G^{b,\Pi^{\mathsf{PKE}}}_{\mathsf{PKE\text{-}CCA}}$ and $\mathsf{Mod}^{b,0}$, SSBee allows us to present a list of SMT-LIB files to express state relations, lemmata, randomness mapping, and custom theories and asssertions for the abstarct functions. Although the keyword `invariant` is used for the list of SMT-LIB files, SSBee does not check their contents and concatenate all files when compiling the final SMT-LIB output. This allows to organize the SMT-LIB files based on their content and reuse them for different oracles. We have benefitted from this feature in the TLS 1.3 key schedule verification. There are three built-in lemmata that should be proved: `invariant`, `same-output`, and `equal-aborts`. We have to highlight that the same-output property defined in Section 2.9 are split into two lemmata `same-output` and `equal-aborts` in SSBee. Lemma `same-output` states that the left and right oracles return the same output but shall not abort (given all other conditions stated in 2.9). Lemma `equal-aborts` requires that the left oracle aborts if and only if the right oracle aborts. Unlike the formalization of Theorem 2.3, SSBee does not consider an oracle abort as a special output. Lemma `invariant` requires to prove invariance of the presented state relations. SSBee allows the user to express a list of lemmata in front of each lemma. These lemmata on the right are assumed when trying to prove the lemma on the left. For example, the built-in lemma `no-abort` states that games do not abort and is assumed when proving the lemma `invariant` and `same-output` but not for `equal-aborts`. Lemma `no-abort` can not be proved as a standalone lemma but can be added as a dependency for any other lemma. For all lemmata, SSBee assumes the state relation holds in the old game states. Therefore, it is not needed to add `invariant` in front of the lemmata. Informally, SSBee tries to prove the following

basic properties:

ssbee-randomness-mapping $\wedge$ invariant(old-states) $\wedge$ no-abort $\Longrightarrow$ invariant(new-states)
ssbee-randomness-mapping $\wedge$ invariant(old-states) $\wedge$ no-abort $\Longrightarrow$ same-output
ssbee-randomness-mapping $\wedge$ invariant(old-states) $\Longrightarrow$ equal-aborts

For each lemma `lemma1: [lemma2, lemma3]`, SSBee proves:

ssbee-randomness-mapping $\wedge$ invariant(old-states) $\wedge$ lemma2 $\wedge$ lemma3 $\Longrightarrow$ lemma1

Formally, these properties are expressed as first order logic formulae in SMT-LIB as follows. We encourage the reader to refer back to the definitions 2.9 and 2.7 of the same-output property and invariant state relations, respectively.

For same-output:

$$\left(\begin{array}{c}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}}) \in I \wedge \\ (y^l, S^l_{\mathsf{new}}) \leftarrow O^l(x, S^l_{\mathsf{old}}) \wedge \\ (y^r, S^r_{\mathsf{new}}) \leftarrow O^r(x, S^r_{\mathsf{old}}) \wedge \\ y^l \neq \mathsf{abort} \wedge y^r \neq \mathsf{abort} \wedge \\ \text{ssbee-randomness-mapping}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}})\end{array}\right) \Longrightarrow y^l = y^r$$

For equal-aborts:

$$\left(\begin{array}{c}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}}) \in I \wedge \\ (y^l, S^l_{\mathsf{new}}) \leftarrow O^l(x, S^l_{\mathsf{old}}) \wedge \\ (y^r, S^r_{\mathsf{new}}) \leftarrow O^r(x, S^r_{\mathsf{old}}) \wedge \\ \text{ssbee-randomness-mapping}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}})\end{array}\right) \Longrightarrow \Big((y^l = \mathsf{abort}) \Leftrightarrow (y^r = \mathsf{abort})\Big)$$

For invariance:

$$\left(\begin{array}{c}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}}) \in I \wedge \\ (y^l, S^l_{\mathsf{new}}) \leftarrow O^l(x, S^l_{\mathsf{old}}) \wedge \\ (y^r, S^r_{\mathsf{new}}) \leftarrow O^r(x, S^r_{\mathsf{old}}) \wedge \\ y^l \neq \mathsf{abort} \wedge y^r \neq \mathsf{abort} \wedge \\ \text{ssbee-randomness-mapping}(S^l_{\mathsf{old}}, S^r_{\mathsf{old}})\end{array}\right) \Longrightarrow (S^l_{\mathsf{new}}, S^r_{\mathsf{new}}) \in I$$

SSBee feeds these SMT-LIB formulae to its backend SMT solver (currently CVC5) one by one for each lemma and reports the SMT solver output. If it receives *unsat*, it reports the property is proved and proceeds to the next lemma. If it receives *unknown*, it stops and reports to the user that the proof has failed. Only if it receives *sat*, it outputs the model as a counterexample generated by the SMT solver.

2.6.3 Randomness mapping

The major difference of the three proof obligations in previous section with the definitions 2.9 and 2.7 are the randomness mapping. Although the formalization of Theorem 2.3 makes the randomness explicit and add the randomness string as

an argument to the oracles and adversary, SSBee chooses a different approach without adding extra arguments to the adversaries. Conceptually, SSBee considers a randomness tape for each randomness sampling operation in the game. SSBee then stores (in the state of the game) an integer counter *ctr* as the tape index of the next randomness string string to be consumed from the randomness tape for each of the sampling operations in the game. When a sampling occurs, the counter is incremented as if the tape pointer advances to the next index. When proving equivalence of two games, SSBee assigns a unique integer identifier *id* (in each game) to each sampling operation `<-$` in the code. (One can think of it as the identifier of the randomness tape.) For example, in the game `MonolithicPkeCcaGame`, SSBee assigns 0 to the sampling `r <-$ Bits(2000)`, 1 to the `r <-$ Bits(3000)` in package `KemScheme`, and 2 to the `r <-$ Bits (500)` in package `DemScheme`. However, in game `ModularPkeCcaGame`, SSBee assigns 0 to the sampling `k1 <-$ Bits(256)` in package in `Key`, 1 to the `r <-$ Bits(500)` in package `DemScheme`, 2 to the `r <-$ Bits(2000)`, and 3 to the `r <-$ Bits(3000)` in package `KemScheme`. Following pseudocodes are generated as part of the LaTeX export of SSBee for the game `MonolithicPkeCcaGame`. Randomness sampling identifiers are marked over arrows.

`pkg_KemScheme`

KEM_GEN()

$r \xleftarrow{0}\$ \{0,1\}^{2000}$

return $kem\_gen(r)$

`pkg_DemScheme`

DEM_ENC(k, m)

$r \xleftarrow{2}\$ \{0,1\}^{500}$

return $dem\_enc(r, k, m)$

KEM_ENCAPS(pk)

$r \xleftarrow{1}\$ \{0,1\}^{3000}$

$k\_ek \leftarrow kem\_encaps(r, pk)$

parse k_ek **as** $\left(k, ek \right)$

return $\left(k, ek, r \right)$

For the game `ModularPkeCcaGame`:

`pkg_Key`

SET(kp)

assert $k = \bot$
if b **then**
  $k1 \xleftarrow{0}\$ \{0,1\}^{256}$
  $k \leftarrow k1$
else
  $k \leftarrow kp$

`pkg_DemScheme`

DEM_ENC(k, m)

$r \xleftarrow{1}\$ \{0,1\}^{500}$
return $dem\_enc(r, k, m)$

`pkg_KemScheme`

KEM_GEN()

$r \xleftarrow{2}\$ \{0,1\}^{2000}$
return $kem\_gen(r)$

KEM_ENCAPS(pk)

$r \xleftarrow{3}\$ \{0,1\}^{3000}$
$k\_ek \leftarrow kem\_encaps(r, pk)$
parse k_ek **as** $(\ k, ek\ )$
return $(\ k, ek, r\ )$

In the next step, when SSBee translates a randomness sampling operation such as `r <-$ Bits(2000)`, it replaces the operation with sample(id, ctr_{id}) where sample is an abstract function to be defined by *ssbee-randomness-mapping* and ctr_{id} is the tape index of tape id stored in the game state.

Since the sampled values are outputs of uninterpreted abstract functions, without the randomness mapping they can have any value, modeling a random string. Therefore, SSBee asks the user to provide a randomness mapping function for each oracle exposed to the adversary. A randomness mapping states which sampling operations from the left and right are assumed to sample the same values from their mapped randomness tapes. The following listing shows the randomness sampling we have defined for the KEM-DEM example:

```
(define-fun randomness-mapping-PKGEN
    (
        (sample-ctr-old-monolithic_pke_cca_game Int)
        (sample-ctr-old-modular_pke_cca_game_with_real_kem Int)
        (sample-id-monolithic_pke_cca_game Int)
        (sample-id-modular_pke_cca_game_with_real_kem Int)
        (sample-ctr-monolithic_pke_cca_game Int)
        (sample-ctr-modular_pke_cca_game_with_real_kem Int)
    )
    Bool
    (or
        (and
            (= sample-ctr-monolithic_pke_cca_game sample-ctr-old-monolithic_pke_cca_game)
            (= sample-ctr-modular_pke_cca_game_with_real_kem sample-ctr-old-
    modular_pke_cca_game_with_real_kem)
            (= sample-id-monolithic_pke_cca_game 0)
            (= sample-id-modular_pke_cca_game_with_real_kem 2)
        )
    )
)

```

```
(define-fun randomness-mapping-PKENC
    (
        (sample-ctr-old-monolithic_pke_cca_game Int)
        (sample-ctr-old-modular_pke_cca_game_with_real_kem Int)
        (sample-id-monolithic_pke_cca_game Int)
        (sample-id-modular_pke_cca_game_with_real_kem Int)
        (sample-ctr-monolithic_pke_cca_game Int)
        (sample-ctr-modular_pke_cca_game_with_real_kem Int)
    )
    Bool
    (or
        (and
            (= sample-ctr-monolithic_pke_cca_game sample-ctr-old-monolithic_pke_cca_game)
            (= sample-ctr-modular_pke_cca_game_with_real_kem sample-ctr-old-
    modular_pke_cca_game_with_real_kem)
            (= sample-id-monolithic_pke_cca_game 2)
            (= sample-id-modular_pke_cca_game_with_real_kem 1)
        )
        (and
            (= sample-ctr-monolithic_pke_cca_game sample-ctr-old-monolithic_pke_cca_game)
            (= sample-ctr-modular_pke_cca_game_with_real_kem sample-ctr-old-
    modular_pke_cca_game_with_real_kem)
            (= sample-id-monolithic_pke_cca_game 1)
            (= sample-id-modular_pke_cca_game_with_real_kem 3)
        )
    )
)
```

Observe how the corresponding sampling operations from the left and right are mapped to each other (e.g. Sampling 0 from the left to sampling 2 on the right).

Now we can present the definition of *ssbee-randomness-mapping* used by the proof obligations:

$$\begin{aligned}&\text{ssbee-randomness-mapping}(S^l_{\text{old}}, S^r_{\text{old}}) && \text{(ssbee-randomness-mapping)}\\ &\quad := \forall id_{\text{left}}, id_{\text{right}}, ctr_{\text{left}}, ctr_{\text{right}}\Big(\\ &\qquad \text{randomness-mapping-ORACLE}(ctr_{id_{\text{left}}}(S^l_{\text{old}}), ctr_{id_{\text{right}}}(S^r_{\text{old}}), id_{\text{left}}, id_{\text{right}}, ctr_{\text{left}}, ctr_{\text{right}})\\ &\qquad\quad \Longrightarrow \mathsf{sample}(id_{\text{left}}, ctr_{\text{left}}) = \mathsf{sample}(id_{\text{right}}, ctr_{\text{right}})\Big)\end{aligned}$$

where $ctr_{id_{\text{left}}}(S^l_{\text{old}})$ and $ctr_{id_{\text{right}}}(S^r_{\text{old}})$ are the randomness tape indices stored in the game state at the beginning of the oracle call. In other words, the mapping only happens for the cases the user has specified in the function *randomness-mapping-ORACLE*. The distinction of the base tape index $ctr_{id_{\text{left}}}(S^l_{\text{old}})$ at the beginning of the oracle call from the quantified tape index ctr_{left} allows the user to specify randomness mappings even when two values are sampled from the same sampling operation. This can happen, for example, if the sampling operation lives in an oracle that is called twice.

2.6.4 Invariants

We finally present our state relations in file `invariant.smt2` referenced by the Listing 1. SSBee requires us to define a function called `invariant` and express all the state relations over the given game states. One can easily determine the selector functions that can be applied to the game states to extract states of the packages in the games.

```
(define-fun invariant
    (
        (state-left <GameState_MonolithicPkeCcaGame_<$<!b!>$>>) ; left
        (state-right <GameState_ModularPkeCcaGame_<$<!false!><!b!>$>>) ; right
    )
    Bool
    (let
        (
            (left_pk (<pkg-state-MON_CCA-<$<!b!>$>-pk> (<game-MonolithicPkeCcaGame-<$<!b
    !>$>-pkgstate-pkg_MON_CCA> state-left)))
            (left_sk (<pkg-state-MON_CCA-<$<!b!>$>-sk> (<game-MonolithicPkeCcaGame-<$<!b
    !>$>-pkgstate-pkg_MON_CCA> state-left)))
            (right_pk_mod_cca (<pkg-state-MOD_CCA-<$$>-pk> (<game-ModularPkeCcaGame-<$<!
    false!><!b!>$>-pkgstate-pkg_MOD_CCA> state-right)))
            (right_pk_kem (<pkg-state-KEM-<$$>-pk> (<game-ModularPkeCcaGame-<$<!false!><!
    b!>$>-pkgstate-pkg_KEM> state-right)))
            (left_c (<pkg-state-MON_CCA-<$<!b!>$>-c> (<game-MonolithicPkeCcaGame-<$<!b!>$
    >-pkgstate-pkg_MON_CCA> state-left)))
            (right_c (<pkg-state-MOD_CCA-<$$>-c> (<game-ModularPkeCcaGame-<$<!false!><!b
    !>$>-pkgstate-pkg_MOD_CCA> state-right)))
            (right_kem_ek (<pkg-state-KEM-<$$>-ek> (<game-ModularPkeCcaGame-<$<!false!><!
    b!>$>-pkgstate-pkg_KEM> state-right)))
            (right_mod_cca_ek (<pkg-state-MOD_CCA-<$$>-ek> (<game-ModularPkeCcaGame-<$<!
    false!><!b!>$>-pkgstate-pkg_MOD_CCA> state-right)))
            (right_dem_c (<pkg-state-MOD_CCA-<$$>-em> (<game-ModularPkeCcaGame-<$<!false
    !><!b!>$>-pkgstate-pkg_MOD_CCA> state-right)))
            (right_key_k (<pkg-state-Key-<$<!key_idealization!>$>-k> (<game-
    ModularPkeCcaGame-<$<!false!><!b!>$>-pkgstate-pkg_Key> state-right)))
            (right_sk (<pkg-state-KEM-<$$>-sk> (<game-ModularPkeCcaGame-<$<!false!><!b!>$
    >-pkgstate-pkg_KEM> state-right)))
            (right_encaps_randomness (<pkg-state-KEM-<$$>-encaps_randomness> (<game-
    ModularPkeCcaGame-<$<!false!><!b!>$>-pkgstate-pkg_KEM> state-right)))
            (right_T (<pkg-state-DEM-<$<!dem_idealization!>$>-T> (<game-ModularPkeCcaGame
    -<$<!false!><!b!>$>-pkgstate-pkg_DEM> state-right)))
        )
        (and
            (= left_pk right_pk_mod_cca right_pk_kem) ; (1)
            (= ((_ is mk-none) left_pk) ((_ is mk-none) left_sk) ((_ is mk-none)
    right_pk_mod_cca) ((_ is mk-none) right_pk_kem) ((_ is mk-none) right_sk)) ; (1) and
    (2)
            (= left_c right_c) ; (3)
            (= ((_ is mk-none) left_c) ((_ is mk-none) right_c) ((_ is mk-none)
    right_kem_ek) ((_ is mk-none) right_mod_cca_ek) ((_ is mk-none) right_dem_c) ((_ is
    mk-none) right_key_k)) (4)
            (= left_sk right_sk) ; (1)
            (= right_mod_cca_ek right_kem_ek) ; (*)
            (=> ((_ is mk-none) right_pk_kem) ((_ is mk-none) right_c)) ; (4)
```

```
            (=> ; (7)
                (not ((_ is mk-none) right_c))
                (= (maybe-get right_c) (mk-tuple2 (maybe-get right_mod_cca_ek) (maybe-get
     right_dem_c)))
            )
            (=> ; (**)
                (not ((_ is mk-none) right_key_k))
                (and
                    (= (maybe-get right_key_k) (el2-1 (<<func-proof-kem_encaps>> (maybe-
    get right_encaps_randomness) (maybe-get right_pk_kem))))
                    (= (maybe-get right_kem_ek) (el2-2 (<<func-proof-kem_encaps>> (maybe-
    get right_encaps_randomness) (maybe-get right_pk_kem))))
                )
            )
            (forall ; (5)
                (
                    (x Bits_*)
                )
                (and
                    (=>
                        ((_ is mk-none) right_c)
                        ((_ is mk-none) (select right_T x))
                    )
                    (=>
                        (not ((_ is mk-none) right_c))
                        (= (= x (maybe-get right_dem_c)) (not ((_ is mk-none) (select
    right_T x))))
                    )
                )
            )
        )
    )
)
```

We have annotated each state relation that corresponds to a state relation in Definition 2.13. (Notice that left and right games are swapped.) The state relation `(*)` uses the ghost state `ek` of the package `MOD_CCA`. Moreover, state relation `(**)` uses the ghost state `encaps_randomness`.

We want to emphasize that the same state relations and lemmata are used to verify equivalence game hops $G^{0,\Pi^{\text{PKE}}}_{\text{PKE-CCA}} \overset{code}{\equiv} H_1$ and $G^{1,\Pi^{\text{PKE}}}_{\text{PKE-CCA}} \overset{code}{\equiv} H_4$ separately. Refer to the repository [Raj25a] to see the invariant files.

As mentioned before, SSBee, at the time of writing this thesis, does not check the state relation holds in the initial game states. One has to first construct initial states `S0_left` and `S0_right` themselves. (Usually all state variables are none and tables map all entries to none.) In the next step they have to query the SMT solver with `(assert ( not (invariant S0_left S0_right)))`. Notice the negation of `(invariant S0_left S0_right)` is used to prove the invariant holds in the initial states.

2.6.5 KEM scheme correctness property as a lemma

We expressed the correctness property of the KEM scheme, necessary to prove the same-output property of the oracle PKDEC, as a lemma. Initially, we defined the property as an assertion in the followin listing:

```
(assert
    (forall
        (
            (gen-r Bits_2000)
            (encaps-r Bits_3000)
        )
        (let
            (
                (pk (el2-1 (<<func-proof-kem_gen>> gen-r)))
                (sk (el2-2 (<<func-proof-kem_gen>> gen-r)))
            )
            (let
                (
                    (k (el2-1 (<<func-proof-kem_encaps>> encaps-r pk)))
                    (ek (el2-2 (<<func-proof-kem_encaps>> encaps-r pk)))
                )
                (= k (<<func-proof-kem_decaps>> sk ek))
            )
        )
    )
)
```

However, this results in unknown result from the SMT solver. Instead, we helped the SMT solver by adding it as a lemma for the same-output lemma as follows:

```
(define-fun <relation-lemma-kem-correctness-monolithic_pke_cca_game-
    modular_pke_cca_game_with_real_kem-PKDEC>
    (
        (old-state-left <GameState_MonolithicPkeCcaGame_<$<!b!>$>>)
        (old-state-right <GameState_ModularPkeCcaGame_<$<!false!><!b!>$>>)
        (return-left <OracleReturn-MonolithicPkeCcaGame-<$<!b!>$>-MON_CCA-<$<!b!>$>-PKDEC
    >)
        (return-right <OracleReturn-ModularPkeCcaGame-<$<!false!><!b!>$>-MOD_CCA-<$$>-
    PKDEC>)
        (ek_ctxt (Tuple2 Bits_400 Bits_*))
    )
    Bool
    (let
        (
            (pk (<pkg-state-KEM-<$$>-pk> (<game-ModularPkeCcaGame-<$<!false!><!b!>$>-
    pkgstate-pkg_KEM> old-state-right)))
            (sk (<pkg-state-KEM-<$$>-sk> (<game-ModularPkeCcaGame-<$<!false!><!b!>$>-
    pkgstate-pkg_KEM> old-state-right)))
        )
        (=>
            (not ((_ is mk-none) pk))
            (forall
                (
                    (r Bits_3000)
```

```
                )
                (let
                    (
                        (k (el2-1 (<<func-proof-kem_encaps>> r (maybe-get pk))))
                        (ek (el2-2 (<<func-proof-kem_encaps>> r (maybe-get pk))))
                    )
                    (= k (<<func-proof-kem_decaps>> (maybe-get sk) ek))
                )
            )
        )
    )
)
```

Notice that we don't need quantification on the randomness string `gen-r` used to generate the public/secret key pair because we only need to express the property for the already generated key pair in the game state.

2.6.6 Randomness mapping issue

When proving the same-output lemma for the oracle `PKENC`, we needed to map the randomness samplings both in the package `Key` and `DEM` to generate an encryption key and encrypt a message, respectively. Earilier, we presented the randomness mapping function for the oracle `PKENC` which mapped two sampling points. We observed that multiple randomness mappings can be tricky for the SMT solvers as they have difficulty instantiating universal quantifier in equation ssbee-randomness-mapping. Precisely, the function `randomness-mapping-PKENC` specifies mapping of sampling point 2 from the left to sampling point 1 from the right. In a small experiment with CVC5, we asked CVC5 to prove $\mathsf{sample}(2, c) = \mathsf{sample}(1, c')$ or $\mathsf{sample}(1, c) = \mathsf{sample}(3, c')$ when given the function `randomness-mapping-PKENC` and assuming

$$\begin{aligned}\forall id_{\mathsf{left}}, id_{\mathsf{right}}, ctr_{\mathsf{left}}, ctr_{\mathsf{right}} \Big(& \\ & \text{randomness-mapping-PKENC}(c, c', id_{\mathsf{left}}, id_{\mathsf{right}}, ctr_{\mathsf{left}}, ctr_{\mathsf{right}}) \\ & \implies \mathsf{sample}(id_{\mathsf{left}}, ctr_{\mathsf{left}}) = \mathsf{sample}(id_{\mathsf{right}}, ctr_{\mathsf{right}}) \Big)\end{aligned}$$

for arbitrary c and c'. In all runs, we received an unknown result. However, the same obligation was proved quickly with Z3.

To prevent unknown results and ensure fast verification, we help the SMT solver with quantifier instantiation and express the follwoing randomness mappings directly as lemma `lemma-rand-is-eq`.

$$\begin{aligned}\mathsf{sample}(2, ctr_2(S^l_{\mathsf{old}})) &= \mathsf{sample}(1, ctr_1(S^r_{\mathsf{old}})) \\ \mathsf{sample}(1, ctr_1(S^l_{\mathsf{old}})) &= \mathsf{sample}(3, ctr_3(S^r_{\mathsf{old}}))\end{aligned}$$

Nevertheless, we leave more investigation of this issue to a future work. Concretely, we speculate that proper e-matching patterns for the universal quantifier can help the SMT solver to instantiate the quantifier properly.

2.7 How to run SSBee?

At the time of writing this thesis, SSBee is a command line application being actively developed by Chris Brzuska, Christoph Egger, and Jan Winkelmann using Rust programming language. Therefore, to use SSBee, one needs to have a working Rust installation on their system. The next step is moving to the SSBee project directory containing the `ssp.toml` file and running `cargo run -p ssbee prove -t`. The `-t` instructs SSBee to output the SMT-LIB file in the `_build` directory before feeding in into the SMT sovler. The other useful command is `cargo run -p ssbee latex` which exports the packages and all the composition digrams in LaTeX, such as the ones shown in the previous section.

3 TLS 1.3 Key Schedule

In this section, we give an overview of internals of TLS 1.3 handshake, key schedule as well as its security analysis. Although explicitly mentioned, many figures and pseudocodes in this section are adopted from the paper of [BDLE+21], which forms the basis of this thesis.

3.1 TLS 1.3 Handshake and Key Schedule

TLS 1.3 standard [Res18] specifies a key exchange protocol that allows two internet endpoints, a client (initiator) and server, to authenticate each other, agree on a freshly generated secret shared key, and achieve a secure channel to send messages. Key schedule refers to all cryptographic key computation operations that derive the shared secret key among other keys. TLS 1.3 supports three modes of authentication: certificate-based authentication with (Elliptic Curve) Ephemeral Diffie-Hellman (EC)DHE key exchange (`dh_ke`), pre-shared key (PSK) authentication with (EC)DHE key exchange (`psk_dh_ke`), and PSK only authentication without (EC)DHE key exchange (`psk_ke`). The last authentication mode (`psk_ke`) comes with weaker security guarantees, most importantly at the cost of no forward secrecy for application data. Forward secrecy refers to the security notion that current session keys still remain secure even though long term keys (such as PSKs) are compromised in the future, hence the name forward secrecy.

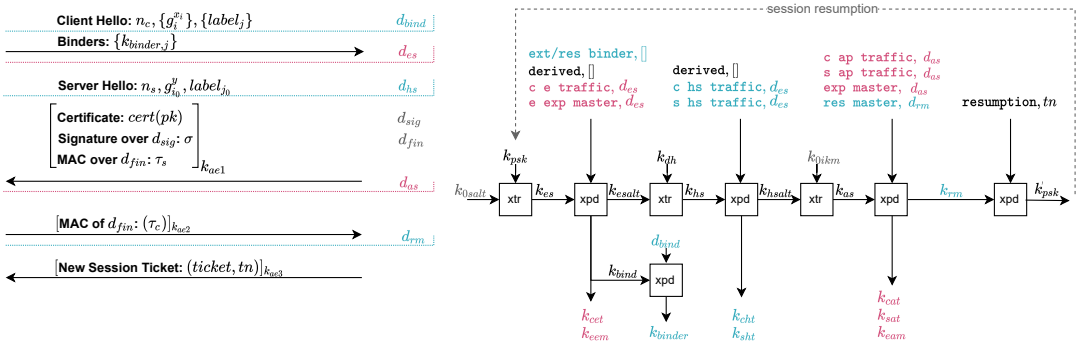
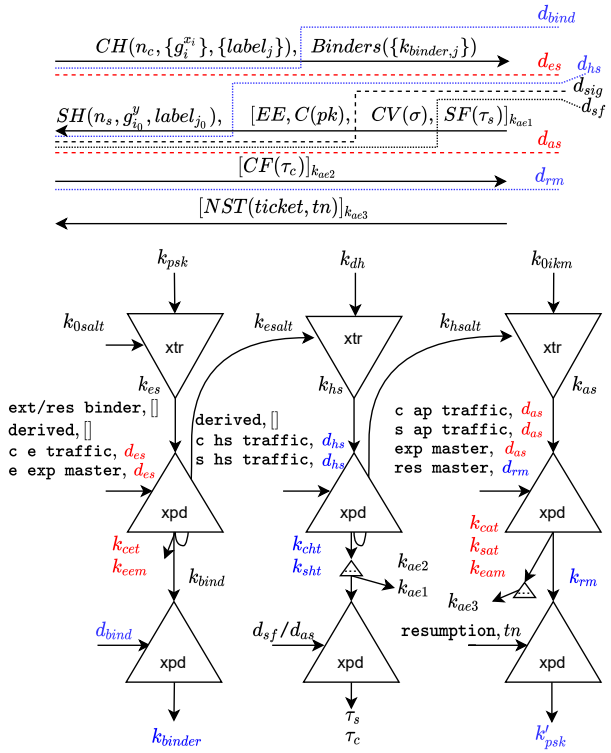


Figure 9: TLS 1.3 message flow on the left and key schedule overview on the right (copied with permission from [BDLE+21]): messages in brackets are encrypted with the key in the subscript

TLS 1.3 handshake protocol comprises of seven main key exchange messages: Client Hello (CH), Server Hello (SH), EncryptedExtensions (EE), Server Certificate $C(pk)$, Server CertificateVerify $CV(\sigma)$, Server Finished (SF), and Client Finished (CF) messages. The first message (CH) aims for cryptographic negotiation as well as sharing random nonces n_c along with Diffie-Hellman (DH) shares $\{g_i^{x_i}\}$ for supported groups with the server. Moreover, in case of PSK-based authentication modes (i.e. `psk_dh_ke` or `psk_ke`), client sends identities $\{label_j\}$ of pre-shared keys, one of which shall be chosen by the server if is is known to them. CH message also contains a



Parent name(s):		
n:	n_1	n_2
es:	*0salt*	*psk*
eem:	*es*	$\perp$
cet:	*es*	$\perp$
bind:	*es*	$\perp$
binder:	*bind*	$\perp$
hs:	*esalt*	*dh*
sht:	*hs*	$\perp$
cht:	*hs*	$\perp$
hsalt:	*hs*	$\perp$
as:	*hsalt*	*0ikm*
rm:	*as*	$\perp$
cat:	*as*	$\perp$
sat:	*as*	$\perp$
eam:	*as*	$\perp$
psk:	*rm*	$\perp$

$k_{es} \leftarrow \mathsf{xtr}(k_{psk}, k_{0salt})$
$k_{bind} \leftarrow \mathsf{xpd}(k_{es}, \texttt{ext/res binder}, [])$
$k_{binder} \leftarrow \mathsf{xpd}(k_{bind}, [], d_{bind})$
$k_{cet} \leftarrow \mathsf{xpd}(k_{es}, \texttt{c e traffic}, d_{es})$
$k_{eem} \leftarrow \mathsf{xpd}(k_{es}, \texttt{e exp master}, d_{es})$
$k_{esalt} \leftarrow \mathsf{xpd}(k_{es}, \texttt{derived}, [])$
$k_{hs} \leftarrow \mathsf{xtr}(k_{esalt}, k_{dh})$
$k_{cht} \leftarrow \mathsf{xpd}(k_{hs}, \texttt{c hs traffic}, d_{hs})$
$k_{sht} \leftarrow \mathsf{xpd}(k_{hs}, \texttt{s hs traffic}, d_{hs})$
$k_{hsalt} \leftarrow \mathsf{xpd}(k_{hs}, \texttt{derived}, [])$

$k_{as} \leftarrow \mathsf{xtr}(k_{hsalt}, k_{0ikm})$
$k_{cat} \leftarrow \mathsf{xpd}(k_{as}, \texttt{c ap traffic}, d_{as})$
$k_{sat} \leftarrow \mathsf{xpd}(k_{as}, \texttt{s ap traffic}, d_{as})$
$k_{eam} \leftarrow \mathsf{xpd}(k_{as}, \texttt{exp master}, d_{as})$
$k_{rm} \leftarrow \mathsf{xpd}(k_{as}, \texttt{res master}, d_{rm})$
$k'_{psk} \leftarrow \mathsf{xpd}(k_{rm}, \texttt{resumption}, tn)$
$k_{psk}, k_{0ikm}, k_{0salt} \leftarrow 0^{\mathsf{len}(alg)}$
$\tau_{s/c} \leftarrow \mathsf{deriveFin}(k_{sht}/k_{cht}, d_{sf}/d_{as})$
$\sigma \leftarrow \mathsf{sign}(sk, d_{sig})$
$k_{ae\{1,2,3\}} \leftarrow$ TLS record layer derivations

$\mathsf{hmac}_{alg}(k, t)$

$ipad \leftarrow 0x36^{\mathsf{blocksize}(alg)}$
$opad \leftarrow 0x5C^{\mathsf{blocksize}(alg)}$
$k^{\dagger} \leftarrow k \| 0^{|ipad|-|k|}$
$inner \leftarrow \mathsf{hash}\text{-}alg(k^{\dagger} \oplus ipad) \| t$
$d \leftarrow \mathsf{hash}\text{-}alg((k^{\dagger} \oplus opad) \| inner)$
return d

$\mathsf{xpd}\text{-}alg(k_1, (label, d))$

if $label = []$:
  return $\mathsf{hmac}_{alg}(k_1, d)$
$t \leftarrow (\mathsf{len}(alg), \texttt{tls13 } \| label, d) \| 0x01$
$k \leftarrow \mathsf{hmac}_{alg}(k_1, t)$
return k

$\mathsf{xtr}\text{-}alg(k_1, k_2)$

$k \leftarrow \mathsf{hmac}_{alg}(k_1, k_2)$
return k

Figure 10: TLS 1.3 key schedule with more details (copied with permission from [BDLE+21]). Notice "xpd" operation with empty label [] (used only for derivation of k_{binder}) is essentially an HMAC operation instead of HKDF-Expand as illustrated in the pseudocode. BDEFKK treat this operation identical to other xpd operations as pseudorandomness and collision resistance assumptions apply both to HMAC and HKDF-Expand. $alg \in \{\texttt{sha256}, \texttt{sha384}, \texttt{sha512}\}$ is the hash algorithm used by HMAC.

binder values $k_{binder,j}$ for each PSK identity $\{label_j\}$ [9] Binder values are computed from PSK, which serves as a proof for client knowledge of PSK to the server and binding the current handshake to PSK. It is noteworthy that cryptographic negotiation helps the client and server to agree on the supported elliptic curve or finite field group, signature algorithms, authenticated-encryption-and-associated-data (AEAD) algorithms, and hash functions, among others. DH shares, PSK identities, and binder values are sent among other "extensions" in the Client Hello. Figure 9 and 10 depicts the cryptographic interesting parts of TLS 1.3 message flow. If the server partially agrees with cryptographic parameters offered by the client, it may send a Hello Retry Request due to insufficient information in the CH message, prompting the client to send another CH message with completed information. If the server is able to negotiate an acceptable set of handshake parameters with the client, it will send the SH message. SH message contains (in plaintext) server's random nonce n_s, server's DH share $g_{i_0}^{y}$ for exactly one of the agreed groups i_0, and in case of `psk_dh_ke` mode, selected PSK identity $label_{j_0}$. At this point, server can compute DH secret $k_{dh} = (g_{i_0}^{x_{i_0}})^y = g_{i_0}^{x_{i_0} y}$. Client can also compute the same DH secret after receiving server's DH share in SH message. In the follow-up messages, server encrypts the certificate $C(pk)$, certificate verify $CV(\sigma)$ message, and Server Finished (SF) message under the record layer encryption key k_{ae1}. Certificate $C(pk)$ contains server's public key pk and a certificate authority's (CA) signature over server public key, Certificate verify $CV(\sigma)$ message contains the signature σ over the hash digest of handshake transcript d_{sig} (marked in Figures 9 and 10) under server's private key. SF message contains the message authentication code (MAC) over digest of handshake transcript d_{fin} for key confirmation. The encryption protects aforementioned messages from passive attackers. The MAC tag τ_s is computed using the Server Finished MAC key. SF MAC key and record layer encryption key k_{ae1} are both derived from the server handshake traffic secret k_{sht}.

As soon as the server sends the SF message, both client and server can derive client/server application traffic secrets k_{cat}, k_{sat}, and exporter (application) master secret k_{eam} from master secret k_{as} and the hash digest of transcript until the end of SF message. Exporter master secret can be used by the higher level protocols or applications as a pseudorandom shared key established between the endpoints for their own purposes.

Finally, client verifies the signature in the SF message, and computes a MAC tag τ_c over digest of handshake transcript d_{as} for key confirmation. The MAC tag is computed using Client Finished MAC key. The tag itself is also encrypted with the record layer key k_{ae2}. The CF MAC key together with k_{ae2} are derived from client handshake traffic secret k_{cht}. Figure 10 illustrates how k_{sht} and k_{cht} are derived by the key schedule using the HKDF-Extract (xtr) and HKDF-Expand (xpd) applied to hash of handshake transcripts and input keys (DH secret k_{dh} and PSK k_{psk}). HKDF-Extract

[9] Binder value is the TLS 1.3 standard terminology. However, in the rest of the thesis and also in analysis of [BDLE+21], binder values are called binder keys. Confusingly enough, TLS 1.3 standard refers to k_{bind} (illustrated in figure 10) as binder keys while we call them bind keys, although we don't refer to them very often.

and HKDF-Expand are HMAC-based key derivation functions introduced by Krawczyk [Kra10] and standardized by IETF [KE10]. HMAC is a message authentication code (MAC) based on a cryptographic hash function [BCK96]. TLS 1.3 supports three hash functions `sha256`, `sha384`, and `sha512` respectively with output lengths 256, 384, and 512 bits to be used by HMAC operations. The choice of hash function is agreed upon during cryptographic negotiation as part of Client Hello and the same hash function is used by all HKDF-Extract and HKDF-Expand in the key schedule. The rationale behind using HKDF functions for a key exchange protocol was pioneered in OPTLS protocol [KW15] which also inspired design of TLS 1.3. Briefly, in the TLS 1.3 key schedule, HKDF-Extract is considered to be a randomness extractor generating a pseudorandom key given two high-entropy inputs (keying material). HKDF-Expand is considered to be psuedorandom function (PRF) computing a pseduorandom value given a label (such as `c hs traffic`) and a hash digest (hash of transcript up to some message). Refer to Figure 10 for the definition of xtr, xpd, and HMAC.

After the handshake, endpoints transmit encrypted application messages using the AEAD algorithms (agreed during the handshake) and encryption keys generated by the key schedule. TLS 1.3 record protocol specifies the format of transmitted messages (including handshake messages). Ironically, handshake protocol concerns three other messages called post-handshake messages sent after the handshake is finished : key and initialization vector updates, post-handshake client authentication, and new session ticket (NST) message. Server sends NST message to the client at some point after the handshake is completed (i.e. when Client Finished message is received by the server). The message includes information (ticket value and ticket nonce) that can be used by the client to derive a new PSK to authenticate the server (and for the server to authenticate the client) in a future session instead of exchanging certificates again. The future session authentication is bound to the current handshake and, hence, called resumed session. Similarly, the new PSK is called resumption PSK. The NST message includes ticket value (label in the diagram), which can be any unique label or identity suitable for future server lookup, ticket nonce, ticket lifetime, among others. Ticket values are used as PSK identities by the client to give a clue to the server of which PSK or PSKs client is willing to use. As illustrated in the third column of Figure 10 or at the very right of Figure 9, resumption PSK can be computed by applying the expander (a PRF) on the ticket nonce sent by the server under the resumption master secret k_{rm} derived from keys and digest of transcript of the current session. In this thesis, we refer to the initial PSK shared out-of-band between the endpoints by external PSK or application PSK. Notice that the binder values in the Client Hello message binds the current handshake (resumed session) to the previous handshake from which the PSK was generated (and transitively to the initial handshake). As a final note, PSK identities may be meaningless unique lookup keys issued by the server to lookup the PSK identity offered by the client in a database stored by the server (stateful server) or rather be an encrypted and authenticated object to restore the state of the server upon client offering the object to the server (stateless server).

Similar to exporter master secret, key schedule generates early exporter master secret k_{eem} from PSK in 0-RTT for higher application layer early encryption purposes (for example in a resumed session). Analogous to application traffic secrets, key

schedule generates client early traffic secret k_{cet} for 0-RTT encrypted application traffic.

In addition to DH secret k_{dh} and PSK k_{psk}, k_{0salt} and k_{0ikm} are fixed zero input keys (salts), i.e. $k_{0salt} = k_{0ikm} = 0$. Although extractor xtr is applied to k_{psk} under the fixed zero salt to derive k_{es} (or to k_{hsalt} and k_{0ikm} to derive k_{as}, see Figure 10), xtr or the underlying HMAC is assumed to be dual pseudorandom function. Moreover, TLS 1.3 standard requires that in `dh_ke` key exchange mode where PSKs are not used, k_{psk} is replaced with zero. Similarly, in `psk_ke` key exchange mode where Diffie-Hellman key exchange is not performed, k_{dh} is again replaced zero. TLS 1.3 requires that all zero bitstrings used in the key schedule (namely k_{0salt}, k_{0ikm}, k_{psk} in `dh_ke` mode, and k_{dh} in `psk_ke` mode) have the same length as the outputs of the hash function used by HMAC. Since HMAC pads its key with zeros to obtain a key of the size of output of hash function (see HMAC construction in the bottom of Figure 10), we consider $k_{0salt} = 0$ to be the single bit zero string while other zero keys are all-zero bitstrings with the same length as the output of the hash function.

In conclusion, given the DH secret k_{psk} and the PSK k_{psk} as input, TLS 1.3 key schedule generates eight output keys $k_{cet}, k_{eem}, k_{binder}, k_{cht}, k_{sht}, k_{cat}, k_{sat}, k_{eam}$ for the use of handshake layer (k_{binder} for Client Hello and k_{cht}, k_{sht} for Client and Server Finished messages), record layer ($k_{cet}, k_{cht}, k_{sht}, k_{cat}, k_{sat}$), and application layer (k_{eem}, k_{eam}). We refer to all other keys (other than output and input keys) as internal keys. [10] Nevertheless, this creates a boundary for the key schedule for the security model to capture pseudorandomness and uniqueness of these eight keys.

3.1.1 Towards a key schedule security model

To motivate for the security model, key schedule shall generate pseudorandom and unique output keys if at least one of the input keying material sources (DH secrets or PSKs) are honestly generated (i.e. unknown to an adversary). For instance, one honesty combination is when DH secret is honest because of an authenticated Diffie-Hellman key exchange but PSK may be dishonest and known to the adversary because of using all-zero bitstring in `dh_ke` mode or a compromised long term PSK (modeling forward secrecy). Another honesty combination may involves a dishonest DH secret because of using all-zero bitstring in `psk_ke` mode but an honest PSK. When both DH secret and PSK are honest and unknown to the adversary, we can capture the security in `psk_dh_ke` mode. Key uniqueness is required for the key exchange security. Essentially, a key exchange protocol aims to achieve distinct keys for distinct sessions. This property translates to distinct and unique output keys derived by the key schedule.

TLS 1.3 key schedule achieves key pseudorandomness by applying a series of HKDF-Extract and HKDF-Expand operations to the input keys (DH secret k_{dh} and

[10] Notice that k_{cht}, k_{sht} are unfortunately used both for encryption of handshake messages and deriving MAC keys. Moreover, k_{sat} is used to derive encryption keys (after key updates) for encryption of both server application traffic as well as the special handshake message New Session Ticket sent after the handshake completion by the server. BDEFKK [BDLE+21] suggest in their paper that higher modularity is enforced to separate the keys used for MAC computation and encryption of handshake messages as well as NST message encryption and application traffic encryption.

PSK k_{psk}). Moreover, it achieves key uniqueness by involving digest of the handshake during key derivations. Let us highlight the importance of transcript digests in the protocol as well as its application in key uniqueness. Transcript digests (including nonces, identities, cryptographic parameters, DH shares, etc.) allow the endpoints to make sure key confirmations and signatures have been freshly generated for the current session with the peer's expected key material, implying liveness of the peer and also preventing man-in-the-middle attacks. More importantly, without the transcript digests and specifically Diffie-Hellman shares themselves and their order, an attacker is able to generate colliding dishonest DH secrets, breaking output key uniqueness. Imagine two endpoints exchange DH shares (g^x, g^y). Without the transcripts, (g^y, g^x) will also have the same DH secret g^{xy}. Even worse, if the adversary is actively intercepting communication of two honest endpoints X and Y who respectively send DH shares g^x and g^y to the adversary, it can force the sessions keys to collide by offering $(g^y)^r$ to X and $(g^x)^r$ to Y for adversarially chosen r. Both sessions end up with the DH secret g^{xyr} (even unknown to the adversary) which can lead to further attacks.

A similar collision attack that breaks output key uniqueness arises from PSKs. As discussed before, TLS 1.3 allows authentication through out-of-band pre-shared keys (external or application PSKs) or resumption PSKs (obtained from a previously authenticated session using New Session Ticket message sent by the server). Imagine a scenario where a malicious server Charlie shares a PSK with an honest server Bob and Bob sends a New Session Ticket message to Charlie so he can resume the session in future with a resumption PSK K. Now consider the malicious server Charlie agrees with (or forces) victim client Alice to share the same resumption PSK K with her as an out-of-band application PSK. Although Bob believes he shares PSK K with Charlie, which is in fact the case, he shares the same key with Alice too, while alice believes she only shares a key with Charlie. Therefore, Charlie can forward encrypted messages from Alice to Bob without modifications and they will be accepted by Bob. (Nonces and PSK identities can also be forwarded and reused by Charlie by resuming his session once again with both Alice and Bob.) This attack is a case of Unknown key-share (UKS) attack, which can lead to unwanted and devastating attacks in higher level protocols based on message contents. Since many applications use TLS for an authenticated and secure channel, it best to prevent UKS attacks at this level rather than delegating to application developers to check for vulnerability. For instance, if Bob is an access control server and Charlie is a compromised server trusted by Alice and successful key agreement leads to unauthorized access granted to Charlie. Refer to [BWM99] for other scenarios and discussion.

TLS 1.3 prevents collision of dishonest application and resumption PSKs by distinguishing these keys with labels `ext binder` or `res binder` when deriving binder key (k_{bind}) which in turn is used to derive binder value k_{binder} sent in the Client Hello message and included in the handshake for all other output key derivations. As a result, Charlie can not simply forward messages from Alice to Bob as Bob does not accept the binder value due to PSK type mismatch in Client Hello. Therefore, all output keys (including k_{binder}) enjoy uniqueness despite the possibility of collision for

dishonest PSKs. [11]

3.2 Key Schedule Security Model

BDEFKK [BDLE+21] model the security of the key schedule as indistinguishability of a real and an ideal game in State Separating Proofs (SSP) framework. The adversary can instruct the real game to generate (1) random honest [12] application PSKs and DH shares/secrets, (2) store dishonest [13] application PSKs or DH shares/secrets, (3) perform key derivations with xpd and xtr operations using both honest and dishonest keys to compute internal and output keys, and (4) retrieve the output key values from the game.

To describe the ideal game, we extend the definition of honesty to internal and output keys and refer to a key (internal or output) as honest if it is derived from at least one honest input keying material (honest DH shares or random application PSKs chosen by the game and unknown to the adversary). In the ideal game, the adversary interacts with a simulator that shall simulate all the aforementioned functionalities in the real game except the the output key retrieval that is replaced by an ideal functionality. This ideal functionality returns randomly chosen keys to the adversary upon derivation of honest output keys (those that were derived from at least one honest input source), modeling key pseudorandomness. Additionally, upon derivation of an output key, the ideal functionality makes sure that the key is unique and distinct from all other keys, hence modeling key uniqueness.

The key schedule security model is an example of simulation-based security definition in SSP where the adversary is interacting with a simulator and an ideal functionality. Interestingly, the simulator shall simulate key derivations of all internal keys without having access to the output keys, generated by the ideal functionality and returned only to the adversary.

The real and ideal games are compositions of several SSP packages, each or some of which provide one or some of the quadruple functionalities exposed to the adversary. We will define the real ($\texttt{Gks}^0$) and ideal ($\texttt{Gks}^1(\mathcal{S})$) security games in Section 3.2.8 and present the pseudocode of SSP packages used in the games in Section 3.2.7. Looking ahead, we sketch the security analysis of BDEFKK in Section 3.3. Notice that the ideal game $\texttt{Gks}^1(\mathcal{S})$ is parameterized by a simulator package $\mathcal{S}$. BDEFKK construct a concrete simulator with the required input and output interface, and only then reduce the indistinguishability of $\texttt{Gks}^0$ and $\texttt{Gks}^1(\mathcal{S})$ to the standard security assumptions for

[11] Interestingly, binder values together with `ext/res` labels were introduced to TLS 1.3 in draft 17 following the post handshake client authentication attack found by [CHSvdM16] in their analysis of draft 10. The attack allowed a malicious server trusted by victim Alice to impersonate Alice to another honest server Bob. The attacker could forward nonces and PSK identities and since there was no binder value in the protocol at the time, client signatures on the handshake during client authentication, only included reused nonces and reused identities without any binding to material (only known to Alice and her peer, Charlie) which made the signature valid for reuse in Charlie-Bob session. However, the attack did not cause a collision of PSKs (between Charlie and Bob or Charlie and Alice) but a failure of "higher" level mechanism (client authentication) on top of session resumption through PSKs.

[12] chosen by the game and only known to the game

[13] adversarially chosen

(dual) pseudorandomness and collision resistance of xtr and xpd functions, among others. Before these sections, we explain key concepts in the security model.

Counter-intuitively, the adversary can not see the actual values for honest keys in the ideal or real game. This restriction comes from the key exchange security modeling where the adversary does not have access to internal key computations but rather may control the input keys and observe the output keys. In order to allow the adversary to perform key derivations without knowing internal keys, game assigns administrative identifiers to the keys called "handles" and only return the handles to the adversary. Therefore, when the adversary instructs the game to generate honest keys (fresh DH shares or random application PSKs), the game returns a handle to the generated key and logs a mapping from the handle to the key value in a private table. When the adversary wants to perform key derivation using the handles, the game looks up the handle in its private log table to extract the actual key values, computes the derived key (internal or output) using xpd and xtr operations, returns a handle to the derived key to the adversary and again logs a mapping from the handle to the derived key. For homogenous key handling, the game also returns a handle for dishonest input, internal and output keys even though the adversary knows the corresponding key values and instructs the game itself to store these keys. See [Koh23] for an example of handle-based security definition for a pseudorandom function (PRF), where the adversary can evaluate the PRF on game-chosen random (honest) keys as well as adversarially-chosen (dishonest) keys, capturing multiple-key PRF evaluation security.

3.2.1 Cryptographic Agility

Recall that HMAC can be used with three hash functions `sha256`, `sha384`, and `sha512` in TLS 1.3. The security model allows the adversary to determine the hash function algorithm that is used for key derivations when instructing the game to generate an honest (or store a dishonest) application PSK as part of the handle data. Therefore, the adversary can perform key derivations with the same application PSK but under different hash functions. However, when the adversary chooses a hash function as part of an application PSK handle, the same hash function is used for derivation of all other keys derived from the same application PSK handle. The security model enforces this restriction by tagging all keys with a hash algorithm and tagging computed keys during key derivation with the same algorithm as the input keys. It then ensures the computed key inherits the same hash algorithm tag as the input keys. Even in the `dh_ke` mode where a zero key is used for key derivation instead of an application PSK, the adversary has the chance to determine the hash function algorithm for key derivation which affects the key length. This property of the security model that supports various hash function algorithms is called cryptographic agility. Recall that output length of each of these three hash algorithms are different. For a hash algorithm $alg \in \{\texttt{sha256}, \texttt{sha384}, \texttt{sha512}\}$, we denote its output length by $\mathsf{len}(alg)$.

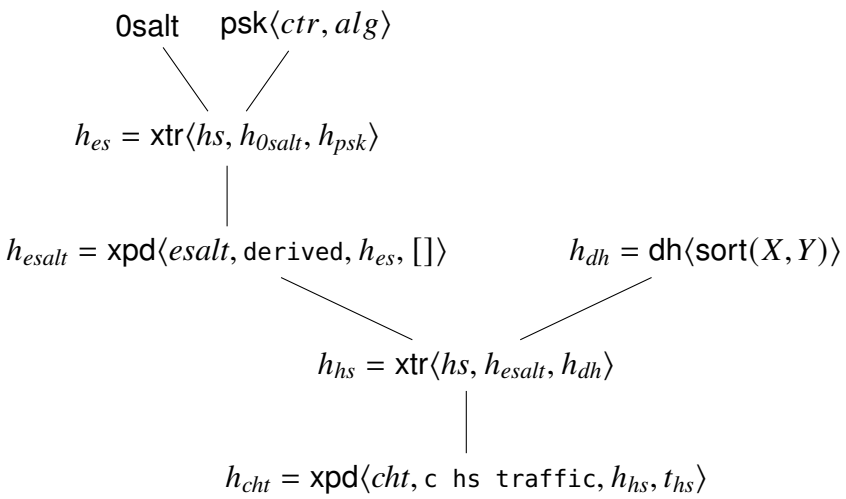
3.2.2 Handles

Key handles are recursive data structures that encompass information, among others, about the key derivation steps and the hash function algorithm tag alg used by HMAC. There are four types of primary base handles corresponding to the four base keys k_{dh}, k_{psk}, k_{0salt}, and k_{0ikm}: $\mathsf{dh}\langle\mathsf{sort}(X,Y)\rangle$ for DH secret g^{xy} constructed from DH shares $X = g^x$ and $Y = g^y$ where $\mathsf{sort}(X,Y)$ lexicographically sorts description of group elements X and Y; $\mathsf{psk}\langle ctr, alg\rangle$ for application PSKs with counter ctr to distinguish different external PSKs and hash algorithm alg chosen by the adversary; $\mathsf{0salt}$ for the fixed zero key $k_{0salt} = 0$; and $\mathsf{0ikm}\langle alg\rangle$ for the fixed zero key $k_{0ikm} = 0^{\mathsf{len}(alg)}$ and adversarially chosen alg. The goal of sorting DH shares X and Y is to have $\mathsf{dh}\langle\mathsf{sort}(X,Y)\rangle = \mathsf{dh}\langle\mathsf{sort}(Y,X)\rangle$ and prevent two distinct handles for the same key g^{xy}. This is critical for the Salted Oracle Diffie Hellman Assumption, one of the assumptions to which the security of key schedule is reduced. Refer to Section 5 for the statement of this assumption and its analysis. To avoid redundancy in the security model, k_{0salt} is a single zero bit because it is used as an HMAC key and HMAC pads keys shorter than its hash function block size with zeros.

With the primary base handles, the security model supports all possible input key combinations in the `psk_dh_ke` mode. To support `psk_ke` and `dh_ke` modes, the model introduces two additional base handles $\mathsf{noDH}\langle alg\rangle$ and $\mathsf{noPSK}\langle alg\rangle$. A $\mathsf{noDH}\langle alg\rangle$ handle can be used in `psk_ke` mode to refer to the absent DH secret replaced by an all-zero bitstring while a $\mathsf{noPSK}\langle alg\rangle$ handle can be used in `dh_ke` mode for the absent application PSK replaced by an all-zero bistring. Recall that TLS 1.3 requires that zero keys have the same length as the output of the hash function. Therefore, $\mathsf{noDH}\langle alg\rangle$ and $\mathsf{noPSK}\langle alg\rangle$ are handles to the keys of form $0^{\mathsf{len}(alg)}$. We consider the hash algorithm tag of the handles and keys to be nullable. Notice that the only keys with empty (null) algorithm tags are k_{dh} and k_{0salt}. Observe that handles of these two keys also have empty (null) algorithm tags.

3.2.3 Handle-based key derivation

The adversary can request key derivations with xtr and xpd operations using the base handles and receive derived handles. These derived handles can again be passed to xtr and xpd to derive further keys, which forms a hierarchy of handles. For instance, the derivation tree below illustrates how a handle to client handshake traffic key k_{cht} is constructed in `psk_dh_ke` mode.



Generally, handles of derived keys have the following structures:

$$\mathsf{xtr}\langle \mathit{name}, \mathit{left\ parent\ handle}, \mathit{right\ parent\ handle}\rangle,$$
$$\mathsf{xpd}\langle \mathit{name}, \mathit{label}, \mathit{parent\ handle}, \mathit{other\ arguments}\rangle.$$

where the *label* is at most 12 characters and used for deriving keys with xpd and *name* is the key name, such as *cht*. *other arguments* is either empty [], the transcript itself (for example t_{hs} in the tree above is the raw transcript before being fed into the hash function to obtain the digest d_{hs}), or ticket nonce (for derivation of resumption PSKs). See Figure 10 for all the labels used for various keys. Observe that except for k_{bind}, there is exactly one label for each key that is derived with an xpd operation. The only exception is the bind key k_{bind}, which is derived with either label `ext binder` or `res binder` depending on whether an application PSK or resumption PSK is used.

As mentioned earlier, key computations and hence handles construct a key (and handle) hierarchy. Observe that each handle (key) has 0, 1, or 2 parents. Handles with zero parents are the base handles and correspond to DH secrets, application PSKs, and fixed salts. Handles with one parent correspond to the keys derived with an xpd operation, a label, and possibly a hash digest of transcript or new session ticket nonce, hence *expand* handles or keys. Handles with two parents corresponds to the keys derived with an xtr operation (i.e. k_{es}, k_{hs}, k_{as}), hence *extract* handle or keys. Notice that handles of application PSKs have no parents while handles of resumption PSKs, i.e. $\mathsf{xpd}\langle \mathit{psk}, \texttt{resumption}, h_{rm}, \mathit{tn}\rangle$, are computed with an xpd operation applied to the handle of a resumption master secret h_{rm}, label `resumption`, and the new session ticket nonce *tn*. See parent name table on the right of Figure 10.

3.2.4 Key names and parents

For ease of referencing, let N be the set of all key names in TLS 1.3, N_{xpd} be the set of all *expand* keys whose last derivation step is an xpd operation, and $N_{\mathsf{xtr}} := \{es, hs, as\}$, set of all *extract* keys whose last derivation step is an xtr operation. Here, we consider that *psk* is derived with xpd from the resumption master secret (as resumption PSK)

instead of a base key; hence, $N_{\mathsf{xpd}} = N \setminus \{es, hs, as, dh, 0salt, 0ikm\}$. Additionally, we define two functions PrntN and Labels to be used in the pseudocode of packages. Let $N_\perp := (N \cup \{\perp\})$ and $\mathsf{PrntN} : N \to N_\perp \times N_\perp$ be a function that returns the parents $(n_1, n_2) = \mathsf{PrntN}(n)$ for each key name n. For instance, $\mathsf{PrntN}(es) = (0salt, psk)$, $\mathsf{PrntN}(psk) = (rm, \perp)$, and $\mathsf{PrntN}(dh) = (\perp, \perp)$. Looking ahead, Figure 11 summarizes all the key names and their parents as a directed graph. Observe that extract keys have two parents, expand keys have exactly one parent, and base keys have no parents. Moreover, let $\mathsf{Labels} : N_{\mathsf{xpd}} \times \{\texttt{ext}, \texttt{res}, \perp\} \to \{0,1\}^{96}$ be a function that returns the 12-character ($12 \times 8 = 96$ bits) label $\mathsf{Labels}(n, r)$ used for deriving expand key $n \in N_{\mathsf{xpd}}$. We use $r = \perp$ for all $n \neq bind$ and $r \in \{\texttt{ext}, \texttt{res}\}$ for $n = bind$. For example, $\mathsf{Labels}(cht, \perp) = \texttt{c hs traffic}$, $\mathsf{Labels}(bind, \texttt{ext}) = \texttt{ext binder}$, and $\mathsf{Labels}(bind, \texttt{res}) = \texttt{res binder}$.

3.2.5 Agile handles

Recall that all base handles except h_{0salt} include the information about the hash algorithm alg used by HMAC. Notice we can still refer to hash algorithm of a derived handle (expand or extract handle) because it inherits and includes the algorithm tag from its root base keys deep inside its recursive structure. Upon each xtr operation, the security model ensures that the algorithm tags of the given handles are all the same in order to make the algorithm tag of all base handles of any constructed handle consistent. As a result, hash algorithm of any (base, extract, expand) handle h is well-defined and is denoted by $\mathsf{alg}(h)$.

3.2.6 Resumption levels

Finally, we define handle resumption levels. Recall that PSKs generated with the resumption master secret and the ticket nonce in a TLS session can be used for authentication in a future session with the binder values and as a base key for derivation of other keys in that session. Since handles comprise the information about the key derivation steps, handles of resumption PSKs record the number of session resumptions as part of their recursive structure. We refer to this number of resumptions as the level of the handle h and denote it by $\mathsf{level}(h)$. For instance, handles of all keys derived in the first TLS session s_0 authenticated with an application PSK have level zero. However, handles of the keys, including the resumption PSK, derived in the next session s_1 from the resumption master secret of session s_0 have level one. Similarly, handles of the keys derived in the new session s_2 from the resumption master secret of session s_1 have level two, and so on. Generally, for $\ell \geq 0$, handles of the keys derived in the session s_ℓ from the resumption master secret of session $s_{\ell-1}$ (or application PSK in session s_0) have level ℓ. One can formally define the level of a handle h with the number of expand handles $\mathsf{xpd}\langle psk, \cdots \rangle$ in the body of h recursively h as follows:

$$\mathsf{level}(h) = \begin{cases} \bot & h = \mathsf{0salt}, \mathsf{0ikm}\langle alg\rangle, \mathsf{dh}\langle \mathsf{sort}(X,Y)\rangle \\ 0 & h = \mathsf{psk}\langle ctr, alg\rangle, \mathsf{noPSK}\langle alg\rangle \\ \mathsf{level}(h_1) & h = \mathsf{xtr}\langle n, h_1, h_2\rangle \wedge n \in \{hs, as\} \\ \mathsf{level}(h_2) & h = \mathsf{xtr}\langle n, h_1, h_2\rangle \wedge n = es \\ \mathsf{level}(h_1) & h = \mathsf{xpd}\langle n, label, h_1, args\rangle \wedge n \neq psk \\ 1 + \mathsf{level}(h_1) & h = \mathsf{xpd}\langle psk, \texttt{resumption}, h_1, tn\rangle \end{cases}$$

Therefore, the key name parenting construct a directed graph as illustrated in the Figure 11 with a cycle connecting a resumption master secret (of the previous level) to the resumption PSK (of the next level).

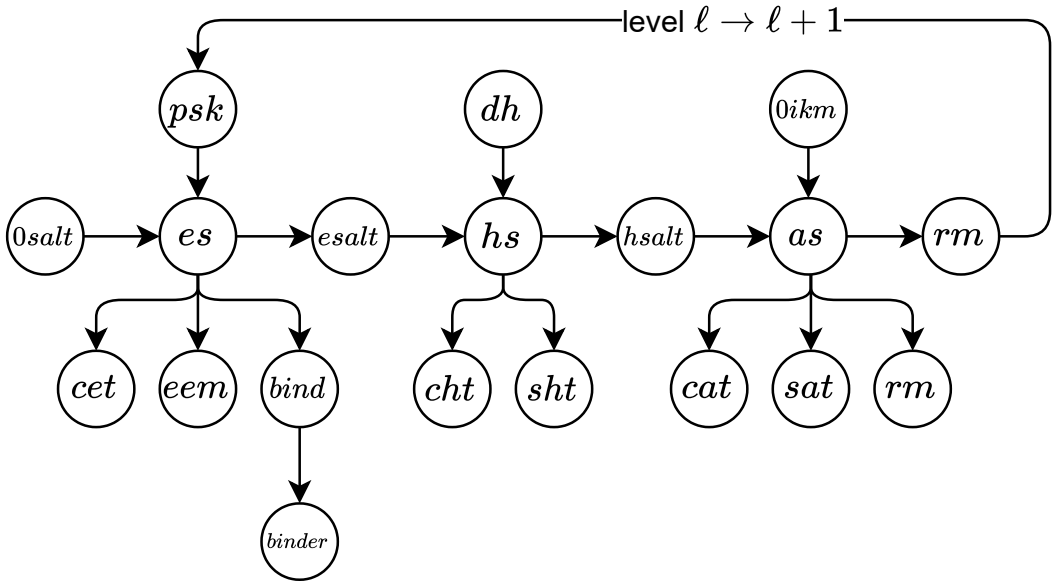


Figure 11: TLS 1.3 key names and parents (copied with permission from [BDLE+21])

In hindsight, handles comprise information about the key derivation steps, hash algorithms, resumption levels, labels and transcripts. Together with the log of handle to key mappings, the security game can compute all the keys in the key schedule.

3.2.7 Packages

In this section, we define SSP packages used in the real ($\texttt{Gks}^0$) and ideal ($\texttt{Gks}^1(\mathcal{S})$) security games. As mentioned in the beginning of Section 3.2, an adversary against the games $\texttt{Gks}^0$ and $\texttt{Gks}^1(\mathcal{S})$ shall instruct the games to (1) generate honest DH shares/secrets and honest application PSKs and receive their handles, (2) set its own dishonest DH shares/secrets and dishonest application PSKs, (3) derive internal and output keys with xtr and xpd operations using the handles, and (4) retrieve the value of output keys. The adversary is allowed to generate as many TLS sessions as it wants and resume these session for at most d many times (i.e. the maximum resumption level of the key handles is d). The games $\texttt{Gks}^0$ and $\texttt{Gks}^1(\mathcal{S})$ expose six types of oracles

for these four functionalities: DHGEN; DHEXP; $\mathsf{SET}_{psk,0}$; $\mathsf{XTR}_{n,\ell}$ for all $n \in N_{\mathsf{xtr}}$ and $\ell \in [d]$; $\mathsf{XPD}_{n,\ell}$ for all $n \in N_{\mathsf{xpd}}$ and $\ell \in [d]$; and $\mathsf{GET}_{n,\ell}$ for all $n \in O^*$ and $\ell \in [d]$, where $[d] := \{0, 1, \ldots, d\}$ and O^* is the set of output keys.

Oracles DHGEN and DHEXP are exposed by package DH. These oracles allow the adversary to generate honest/dishonest DH shares/secrets for all TLS sessions (initial or resumed) it needs. Given a description of the group grp with order $\mathsf{ord}(grp)$, oracle $\mathsf{DHGEN}(grp)$ allows the adversary to generate honest DH shares g^x where g is the generator of the group grp and x is chosen uniformly from the set $Z_{\mathsf{ord}(grp)}$. Oracle $\mathsf{DHEXP}(X, Y)$ allows the adversary to raise any Y to the power of x where $X = g^x$ is generated honestly by DHGEN. The adversary, then, receives the handle $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$ of the key g^{xy}. Similar to the keying package pattern introduced in Section 2.1, oracle DHEXP stores the DH secret g^{xy} in the global (non-leveled) key package $\mathtt{NKey}_{dh}$ via its SET_{dh} oracle. The key package $\mathtt{NKey}_{dh}$ stores the mapping of DH handles h to DH secrets k for the use of other *keyed* packages. We will later on see the pseudocode of package $\mathtt{NKey}_{dh}$. Notice that the oracle DHEXP computes the honesty status hon of the handle $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$ as $hon_X \wedge hon_Y$ and records it also in the key package $\mathtt{NKey}_{dh}$. (The key package $\mathtt{NKey}_{dh}$ stores the honesty bit and the actual key value for each handle.) Therefore, a DH secret g^{xy} would be honest if both of the involved DH shares X and Y are honestly generated by DHGEN. See Figure 12 for the pseudocode of DH package. Clearly, DHGEN enables the adversary to generate honest DH shares and DHEXP allows the adversary to generate honest and dishonest DH secrets using honest and dishonest DH shares, fulfilling the required DH-related functionalities described before.

DH

Parameters

G : set of groups

$\mathsf{ord} : G \to \mathbb{N}$

State

E : table

DHGEN(grp)

assert $grp \in G$

$g \leftarrow \mathsf{gen}(grp)$

$x \leftarrow\!\!\$\ Z_{\mathsf{ord}(grp)}$

$X \leftarrow g^x$

$E[X] \leftarrow x$

return X

DHEXP(X, Y)

assert $\mathsf{grp}(X) = \mathsf{grp}(Y)$

$h \leftarrow \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$

$hon_X \leftarrow E[X] \neq \bot$

$hon_Y \leftarrow E[Y] \neq \bot$

assert $hon_X = 1$

$x \leftarrow E[X]; k \leftarrow Y^x$

$hon \leftarrow hon_X \wedge hon_Y$

$h \leftarrow \mathsf{SET}_{dh}(h, hon, k)$

return h

Figure 12: Package DH

The pseudocode of package DH and other packages in this section are adopted from [BDLE+21]; however, we mark additions with blue and removals with red.

Oracles $\mathsf{XTR}_{n,\ell}$ and $\mathsf{XPD}_{n,\ell}$ are exposed by packages $\mathtt{Xtr}^b_{n,\ell}$ and $\mathtt{Xpd}_{n,\ell}$, respectively. These oracles allow the adversary to derive internal or output key n in a TLS session of level ℓ using the xtr and xpd operations, achieving the third functionality of the security games described before. Figures 13 and 14 show the pseudocode of the packages.

Oracle $\mathsf{XTR}_{n,\ell}(h_1, h_2)$ expects to receive handles (h_1, h_2) of level ℓ with names (n_1, n_2) as parents of the key name n. The oracle verifies its expectation and gets the actual parent key values (k_1, k_2) and honesty bits (hon_1, hon_2) of handles (h_1, h_2) via the oracles $\mathsf{GET}_{n_1,\ell}$ and $\mathsf{GET}_{n_2,\ell}$ exposed by the leveled key packages $\mathtt{Key}^b_{n_1,\ell}$ and

$\mathsf{Key}^b_{n_2,\ell}$. The key package $\mathsf{Key}^b_{n,\ell}$ stores the mapping of handles of level ℓ with name n to the corresponding keys. We will shortly see the pseudocode of the key package $\mathsf{Key}^b_{n,\ell}$. The oracle $\mathsf{XTR}_{n,\ell}(h_1, h_2)$ then derives the key $k_n = \mathsf{xtr}(k_1, k_2)$ and stores k_n in the key package $\mathsf{Key}^b_{n,\ell}$ via its $\mathsf{SET}_{n,\ell}$ oracle. Notice how the package $\mathtt{Xtr}^b_{n,\ell}$ acts as both a *keying* by extracting the parent key values and a *keyed* package by storing the new key value. Finally, the oracle returns the handle $h = \mathsf{xtr}\langle n, h_1, h_2\rangle$ to the key k_n.

The oracle $\mathsf{XTR}_{n,\ell}(h_1, h_2)$ computes the honesty of the output handle h as $hon_1 \vee hon_1$. Therefore, if at least one of the input handles are honest, the output handle will be honest. This reflects the core feature of the key schedule security model that output keys with computed from at least one honest input keying materials should be honest. We will see how the $\mathsf{SET}(h, hon, k^*)$ oracle of the idealized key packages $\mathsf{Key}^1_{n,\ell}$ (in the ideal game) discards the given key k^* and stores a mapping from h to a randomly chosen key k if $hon = 1$, achieving pseudorandomness of honest handles. Notice that the oracle $\mathsf{SET}(h, hon, k^*)$ also records the honesty bit of the given handle h. Therefore, the key packages $\mathsf{Key}^b_{n,\ell}$ store the actual key value as well as the honest bit of handles in level ℓ with name n.

The idealized package $\mathtt{Xtr}^1$ is also seen to sample a uniform random key k^* if the second parent handle is honest. As hinted before, all keys in the security model are tagged with a hash algorithm. We denote the algorithm tag of a key k with $\mathsf{alg}(k)$. BDEFKK show that keys stored in a key package have the same algorithm tag as the handles that map to the keys. Therefore, $\mathsf{alg}(k_1) = \mathsf{alg}(h_1)$ and $\mathsf{alg}(k_2) = \mathsf{alg}(h_1)$. Since $\mathsf{alg}(h_1) = \mathsf{alg}(h_2), \mathsf{alg}(k_1) = \mathsf{alg}(k_2)$. Key derivation function $k = \mathsf{xtr}(k_1, k_2)$ tags k with the well-defined tag $\mathsf{alg}(k) = \mathsf{alg}(k_1) = \mathsf{alg}(k_2)$ inherited from its parents (See the pseudocode of xtr in Figure 20 using the pseudocode of xtr-$alg(k_1, k_2)$ in Figure 10). As a result, the raw random key k^* is tagged with the same tag $\mathsf{alg}(k)$ to obtain the tagged key $\mathsf{tag}_{\mathsf{alg}(k)}(k^\star)$. Looking forward, in the ideal security game, the package $\mathtt{Xtr}^b_{n,\ell}$ is only idealized for $n = hs$ where DH secret k_{dh} is mixed with early salt k_{esalt} to derive handshake salt k_{hs}. The packages $\mathtt{Xtr}^b_{es,\ell}$ and $\mathtt{Xtr}^b_{as,\ell}$ are not idealized in the ideal game. In order to idealize the package $\mathtt{Xtr}^b_{hs,\ell}$ (i.e. switch from $\mathtt{Xtr}^0_{hs,\ell}$ to $\mathtt{Xtr}^1_{hs,\ell}$), BDEFKK reduce to the Salted

$\mathtt{Xtr}^b_{n,\ell}$

Parameters

n : name
ℓ : level
b : bit
$\mathsf{PrntN} : N \rightarrow (N_\perp \times N_\perp)$

State

no state

$\mathsf{XTR}_{n,\ell}(h_1, h_2)$

$n_1, n_2 \leftarrow \mathsf{PrntN}(n)$
if $\mathsf{alg}(h_1) \neq \perp \wedge \mathsf{alg}(h_2) \neq \perp$:
  assert $\mathsf{alg}(h_1) = \mathsf{alg}(h_2)$
$h \leftarrow \mathsf{xtr}\langle n, h_1, h_2\rangle$
$(k_1, hon_1) \leftarrow \mathsf{GET}_{n_1,\ell}(h_1)$
$(k_2, hon_2) \leftarrow \mathsf{GET}_{n_2,\ell}(h_2)$
$k \leftarrow \mathsf{xtr}(k_1, k_2)$
$hon \leftarrow hon_1 \vee hon_2$
if $b \wedge hon_2$:
  $k^\star \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(k)}$
  $k \leftarrow \mathsf{tag}_{\mathsf{alg}(k)}(k^\star)$
$h \leftarrow \mathsf{SET}_{n,\ell}(h, hon, k)$
return h

Figure 13: Package $\mathtt{Xtr}^b_{n,\ell}$

Oracle Diffie-Hellman (SODH) assumption in a game hop.

Oracle $\mathsf{XPD}_{n,\ell}(h_1, r, args)$ is similar to $\mathsf{XTR}_{n,\ell}(h_1, h_2)$ but performs xpd key derivations. The oracle expects to receive a handle h_1 of level ℓ with name n_1 as parent of key name n. Similar to the oracle $\mathsf{XTR}_{n,\ell}$, it verifies this expectation and gets the actual parent key value k_1 and honesty bit *hon* of handle h_1 via the oracle $\mathsf{GET}_{n_1,\ell}$ of the key package $\mathtt{Key}^b_{n_1,\ell}$.

Additionally, it receives a PSK resumption indicator $r \in \{\mathtt{ext}, \mathtt{res}, \bot\}$ as well as other arguments *args* (transcript or ticket nonce), necessary for key derivations. Using the resumption indicator r and key name n, the oracle XPD relies on the function $\mathsf{Labels}(n, r)$ to compute the correct label for key derivations. For $n \neq bind$, the oracle expects $r = \bot$, while for $n = bind$, the oracle expects $r = \mathtt{ext}$ if $\mathsf{level}(h_1) = 0$ and $r = \mathtt{res}$ is $\mathsf{level}(h_1) > 0$. For $n \neq psk$, *args* should be the transcript and $\mathsf{XPD}_{n,\ell}$ computes the hash digest of *args* using the HASH oracle of $\mathtt{Hash}^0$ package. (See package $\mathtt{Hash}^0 = \mathtt{Gacr}^{\mathsf{hash},0}$ in Figure 19) For $n = psk$, *args* should be the ticket nonce *tn* and is not hashed. However, the new computed resumption PSK belongs to the next level. Therefore, ℓ is incremented by one.

Similar to $\mathsf{XTR}_{n,\ell}$, key derivation function $k = \mathsf{xpd}(k_1, (label, d))$ tags k with algorithm $\mathsf{alg}(k_1)$ inherited from k_1 (See the pseudocode of $\mathsf{xpd}(k_1, (label, d))$ in Figure 20 using the pseudocode of xpd-$alg(k_1, (label, d))$ in Figure 10). Therefore, $\mathsf{alg}(k) = \mathsf{alg}(k_1)$. Since BDFKK show $\mathsf{alg}(h_1) = \mathsf{alg}(k_1)$, we conclude $\mathsf{alg}(h) = \mathsf{alg}(k)$ from the handle structure. Finally, $\mathsf{XPD}_{n,\ell}$ returns the handle $h = \mathsf{xpd}\langle n, label, h_1, args\rangle$ and stores the computed key k_n and the honesty bit *hon* inherited from its parent in the key package $\mathtt{Key}^b_{n,\ell}$ under the handle h.

$\mathtt{Xpd}_{n,\ell}$

Parameters

n : name

ℓ : level

$\mathsf{PrntN} : N \to (N_\bot \times N_\bot)$

$\mathsf{Labels} : N \times \{\mathtt{ext}, \mathtt{res}, \bot\} \to \{0,1\}^{96}$

State

no state

$\mathsf{XPD}_{n,\ell}(h_1, r, args)$

$n_1, \_ \leftarrow \mathsf{PrntN}(n)$
$label \leftarrow \mathsf{Labels}(n, r)$
$h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
$(k_1, hon) \leftarrow \mathsf{GET}_{n_1,\ell}(h_1)$
if $n = psk$:
  $\ell \leftarrow \ell + 1$
  $k \leftarrow \mathsf{xpd}(k_1, (label, args))$
else
  $d \leftarrow \mathsf{HASH}(args)$
  $k \leftarrow \mathsf{xpd}(k_1, (label, d))$
$h \leftarrow \mathsf{SET}_{n,\ell}(h, hon, k)$
return h

Figure 14: Package $\mathtt{Xpd}_{n,\ell}$

In order to verify the expectations of the oracle $\mathsf{XPD}_{n,\ell}(h_1, r, args)$ about the resumption indicator r and the transcript *args*, adversary's queries to this oracle are first proxied through package Check.

The package Check exposes oracles $\mathsf{XPD}_{n,\ell}(h_1, r, args)$ for all $n \in N_{\mathsf{xpd}}$ and $\ell \in [d]$ with the pseudocode in Figure 15. The oracle $\mathsf{XPD}_{n,\ell}(h_1, r, args)$ of package Check checks the value of r and transcript content. For $n = bind$, resumption indicator should correspond with the level of handle. Namely, for the initial TLS session (level zero), the binder key k_{bind} is derived with label `ext binder` to indicate usage of an application PSK while for the resumed TLS sessions (non-zero levels), label

`res binder` to indicate usage of a resumption PSK. The other checks correspond to transcript content checks. Let O^* be the set of all output keys (i.e. $O^* = \{cet, eem, cht, sht, cat, sat, eam, rm\}$). For all output keys $n \in O^*$, the oracle ensures the binder value in the transcript (in $args$) is the same as key k_{binder} stored in the key package $\mathsf{Key}^{b}_{binder,\ell}$ under the binder handle $h_{binder} = \mathsf{xpd}\langle binder, [], h_{bind}, t_{bind}\rangle$, where $h_{bind} = \mathsf{xpd}\langle bind, \mathsf{Labels}(bind, r), h_{es}, []\rangle$, t_{bind} is the Client Hello message from $args$, h_{es} is obtained from the recursive structure of h_1 and $r \in \{\mathtt{ext}, \mathtt{res}\}$ depends on the level of h_1. For the output keys $n \in O^* \setminus \{cet, eem\}$ where the transcript includes the DH shares (X, Y) of the client and server, $\mathsf{XPD}_{n,\ell}(h_1, r, args)$ additionally checks the DH handle h_{dh} obtained from the recursive structure of h_1 is exactly $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$.

The package `Check` shows to what extent the key schedule depends on the transcript. Since the choice of arguments (transcript) for xpd operations is under adversarial control, these checks ensure the adversary uses a relevant transcript, those that are generated by the actual runs of the protocol between honest and dishonest parties and adheres to the TLS 1.3 standard key schedule specifications for the format and content of the transcript used by the key derivation functions. As we will see in the security analysis, these key schedule specifications are crucial for the security reduction. Concretely, the checks align with our previous discussion of importance of mixing transcript digests in the key schedule. They prevent the natural Diffie-Hellman key collision and unknown key share (UKS) attack due to external/resumption PSK confusion. Both of these collisions prevent output key uniqueness.

Compared to pseudocode of package `Check` in [BDLE+21], we have added an additional check for resumption indicator r to align with our definition of Labels function. BDEFKK have not clearly defined the Labels function for key names $n \neq bind$.

Moreover, BDEFKK describe their security model and analysis for all *TLS-like key schedule syntaxes*, a generalization of TLS 1.3 key schedule syntax and key parenting graph. In their code of package $\mathtt{Check}_{n,\ell}$, they use $n \in S \cap early$ instead of $n \in \{cet, eem, binder\}$ and $n \in S$ instead of $n \in \{cht, sht, cat, sat, eam\}$, where S is the set of *separation points*. The separation points of a TLS-like key schedule syntax roughly correspond to the xpd operations where the hash digest of the transcript of the protocol is mixed with other keying

`Check`

$\mathsf{XPD}_{n,\ell}(h_1, r, args)$

if $n = bind$:
  if $r = 0$: **assert** $\mathsf{level}(h_1) = 0$
  if $r = 1$: **assert** $\mathsf{level}(h_1) > 0$
elseif $n \in \{cet, eem, binder\}$:
  assert $r = \bot$
  $binder \leftarrow \mathsf{BinderArgs}(args)$
  $h_{binder} \leftarrow \mathsf{BinderHand}(h_1, args)$
  $(k, \_) \leftarrow \mathsf{GET}_{binder,\ell}(h_{binder})$
  assert $binder = k$
elseif $n \in \{cht, sht, cat, sat, eam\}$:
  assert $r = \bot$
  $X, Y \leftarrow \mathsf{DhArgs}(args)$
  $h_{dh} \leftarrow \mathsf{DhHand}(h_1)$
  assert $h_{dh} = \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$
  $binder \leftarrow \mathsf{BinderArgs}(args)$
  $h_{binder} \leftarrow \mathsf{BinderHand}(h_1, args)$
  $(k, \_) \leftarrow \mathsf{GET}_{binder,\ell}(h_{binder})$
  assert $binder = k$
$h \leftarrow \mathsf{XPD}_{n,\ell}(h_1, r, args)$
return h

Figure 15: Package `Check`

material as an input to xpd. Concretely, for TLS 1.3, all the output keys O^* are the separation points (i.e. $S = O^*$) and $\{cet, eem, binder\}$ are the *early* separation points. In this thesis, we focus on the analysis of BDEFKK concretized for TLS 1.3 and leave verification of the analysis for TLS-like key schedule syntaxes to a future work.

Finally, oracles $\mathsf{GET}_{n,\ell}$ for $n \in O^*$ and $\mathsf{SET}_{psk,0}$ are exposed by the packages $\mathsf{Key}^b_{n,\ell}$ and $\mathsf{Key}^b_{psk,0}$, respectively. Figure 16 show the code of $\mathsf{Key}^b_{n,\ell}$ for all $n \in N \setminus \{dh, 0salt, 0ikm\}$ and $\ell \in [d]$.

The oracle $\mathsf{SET}_{n,\ell}$ ensures the name and level of the handle h matches the name and level of the package. It also ensures the algorithm tag of the handle is the same as the key k^* to be stored as well as the length of the raw key [14] is the same as the output length of the hash algorithm. (Recall that $\mathsf{len}(h) = \mathsf{len}(\mathsf{alg}(h))$.) $\mathsf{SET}_{n,\ell}$ queries the oracle Q_n from the package $\mathtt{Log}^{P,map}_n$ to determine whether the key has been set before. As hinted before, observe that the ideal key package $\mathsf{Key}^1_{n,\ell}$ (in the ideal game) discards the given key k^* and samples and stores a random key when $hon = 1$. $\mathsf{SET}_{n,\ell}$ queries the oracle UNQ_n from the package $\mathtt{Log}^{P,map}_n$ to check for uniqueness of the key. We will describe different uniqueness functionalities of the $\mathtt{Log}^{P,map}_n$ depending on its pattern P and mapping map parameters. Finally, $\mathsf{SET}_{n,\ell}$ stores the untagged key and its honesty bit in the table $K_{n,\ell}$ with the index of handle h. Notice that no key is stored in the table when UNQ_n returns a different handle $h' \neq h$ (if a uniqueness mapping occurs).

The oracle $\mathsf{GET}_{n,\ell}$ ensures there is a key in the table with handle h and implicitly matches the name and level of the handle with the package parameters because only such matching handles and keys are stored in the table by $\mathsf{SET}_{n,\ell}$. The oracle $\mathsf{GET}_{n,\ell}$ then tags the key in the table with the algorithm of the handle and returns both the tagged key and the honesty status of the handle.

Similar to BDEFKK, we initialize the table $K_{psk,0}$ with zero keys $0^{\mathsf{len}(alg)}$ for dishonest handles $\mathsf{noPSK}\langle alg\rangle$. Namely, for $alg \in \mathcal{H}$, we set $K_{psk,0}[\mathsf{noPSK}\langle alg\rangle] = (0^{\mathsf{len}(alg)}, 0)$. This initialization mandates another initialization in the package $\mathtt{Log}^{P,map}_{psk}$ that we will discuss later on.

Before moving to the package $\mathtt{Log}^{P,map}_n$, we introduce

$\mathsf{Key}^b_{n,\ell}$

State

$K_{n,\ell}$: table

$\mathsf{SET}_{n,\ell}(h, hon, k^\star)$

assert $\mathsf{name}(h) = n$
assert $\mathsf{level}(h) = \ell$
assert $\mathsf{alg}(k^\star) = \mathsf{alg}(h)$
$k \leftarrow \mathsf{untag}(k^\star)$
assert $\mathsf{len}(h) = |k|$
if $\mathsf{Q}_n(h) \neq \perp$:
  return $\mathsf{Q}_n(h)$
if b :
  if hon :
    $k \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(h)}$
$h' \leftarrow \mathsf{UNQ}_n(h, hon, k)$
if $h' \neq h$:
  return h'
$K_{n,\ell}[h] \leftarrow (k, hon)$
return h

$\mathsf{GET}_{n,\ell}(h)$

assert $K_{n,\ell}[h] \neq \perp$
$(k^*, hon) \leftarrow K_{n,\ell}[h]$
$k \leftarrow \mathsf{tag}_h(k^*)$
return (k, hon)

Figure 16: Package $\mathsf{Key}^b_{n,\ell}$

[14] Notice that the length of the raw key is independent of the algorithm tag of the key. Essentially, tagged keys k can be viewed as tuples (k^*, alg) where k^* is the raw bitstring and alg is the hash algorithm tag.

the non-leveled key package $\texttt{NKey}_n$ for $n \in \{dh, 0salt, 0ikm\}$. Package $\texttt{NKey}_n$ exposes getter and setter oracles $\mathsf{GET}_{n,\ell\in[d]}$ and SET_n for the DH keys and zero salts. The setter oracle SET_{dh} is called by the oracle DHEXP while the getter oracle $\mathsf{GET}_{n,\ell\in[d]}$ is called by the oracle XTR.

Figure 17 shows the code of this package. Unlike $\texttt{Key}^b_{n,\ell}$, for each key name $n \in \{dh, 0salt, 0ikm\}$, there is only one global key package $\texttt{NKey}_n$. This allows the adversary to use Diffie-Hellman secrets and zero salts across different levels. Recall that DH, 0salt and 0ikm handles do not have levels. The SET_n oracle (only exposed by $\texttt{NKey}_{dh}$) is also lightweight and performs fewer checks because DH secrets do not have algorithm tags. The oracle SET_{dh} calls oracles Q_{dh} and UNQ_{dh} from the package $\texttt{Log}^{P,map}_{dh}$. For ease of notation in `Xtr` and `Xpd` packages, $\texttt{NKey}_n$ exposes oracles $\mathsf{GET}_{n,\ell}$ for all $\ell \in [d]$ although the code for all oracles are the same and level ℓ is ignored. Keys are tagged with $\perp$ as only DH secrets reach to this point and DH handles do not have algorithm tags. Notice that the package does not have idealization parameter and the idealization of DH secrets occur indirectly as part of the Salted Oracle Diffie Hellman (SODH) assumption. Finally, BDEFKK use state initialization for the zero keys while we take care of initial state without storing zero values in the table. Moreover, TLS 1.3 standard specifies that "implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, . . . " (Section 7.4.2 of [Res18]). This property is reflected with the blue line **throw** *abort* in the code of package `NKey`. We found this missing check from the security modeling of BDEFKK during the verification discussed in Section 4. However, the standard only specifies this condition for elliptic curve Diffie-Hellman calculations but not for finite field DH computations. Hence, The alternative solution will be an agile $\texttt{NKey}_{dh}$ package, which similar to the $\texttt{Key}_{psk,0}$ package initializes the key table as $K_{dh}[\mathsf{noDH}\langle alg\rangle] = (0^{\mathsf{len}(alg)}, 0)$ with all-zero keys for dishonest handles $\mathsf{noDH}\langle alg\rangle$. This initialization again mandates another initialization in the package $\texttt{Log}^{P,map}_{dh}$ that we will discuss later on. Notice that in the alternative solution, agile $\texttt{NKey}_{dh}$ package disallows setting zero DH secrets for only elliptic curve groups. Since the information about the group is included in the transcript as part of the cryptographic negotiation messages, the `Check` package has to match the group tag of DH handles with transcript. One may also not bother and allow setting zero keys for all groups. In any case, the `Check` package has to match the transcript with the handle used for key derivations (i.e. if no DH shares are used in the transcript, $\mathsf{noDH}\langle alg\rangle$ handle should be used and if DH shares (X, Y) are used, the group tag and the shares should match the shares in the DH handle $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$). These checks are crucial for the security analysis to prevent collision attacks discussed in Section 3.1.1.

We now look into the code of package $\texttt{Log}^{P,map}_{n}$. Figure 18 shows the pseudocode of the package. Red lines are removed from the original formulation of BDEFKK and blue lines are added. The new Notice that only one package exists for each key name n (i.e. the same package is called by all leveled packages $\texttt{Key}^b_{n,\ell}$ for all levels ℓ). As a result, uniqueness checks for a key name n happen across all levels. To answer the uniqueness queries, the package has one table Log_n for its state that stores triples of the form (h, hon, k) where h is a handle, hon is the honesty bit and k is an untagged

$\mathsf{Log}_n^{P,map}$

Parameters

n : name
P : pattern
map : mapping

State

Log_n : Log table
J_n : Key flag table

$\mathsf{Q}_n(h)$

if $Log_n[h] = \bot$ **then**
  return $\bot$
else $(h', \_, \_) \leftarrow Log_n[h]$
  return h'

$\mathsf{UNQ}_n(h, hon, k)$

if $\Big(\exists\, h' : \mathrm{Log}_n[h'] = (h', hon', k)$
  $\wedge\ \mathsf{level}(h) = r \wedge \mathsf{level}(h') = r'$
  $\wedge\ map(r, hon, r', hon', J_n[k])\Big)$:
  $\mathrm{Log}_n[h] \leftarrow (h', hon, k)$
  $\mathrm{J}_n[k] \leftarrow 1$
  return h'
if $\Big(\exists\, h^\star : \mathrm{Log}_n[h^\star] = (h'', hon', k)$
  $\wedge\ \mathsf{level}(h) = r \wedge \mathsf{level}(h^\star) = r'$:
  $\wedge\ P_{cond}(r, hon, r', hon')\Big)$:
  $P_{cmd}(r, hon, r', hon')$
if $P = R \wedge \exists\, h^\star : \mathrm{Log}_n[h^\star] = (h'', hon', k)$:
  throw *win*
$Log_n[h] \leftarrow (h, hon, k)$
return h

map	the condition $map(r, hon, r', hon', \mathrm{J}_n[k])$ is
0	0
1	$hon = hon' = 0 \wedge$ $\wedge\ r \neq r' \wedge 0 \in \{r, r'\} \wedge \mathrm{J}_n[k] \neq 1$
∞	$hon = hon' = 0$

P	the condition $P_{cond}(r, hon, r', hon')$ is	the command $P_{cmd}(r, hon, r', hon')$ is
Z	$\emptyset$	$\emptyset$
A	$hon = hon' = 0 \wedge r = r' = 0$	**throw** *abort*
D	$hon = hon' = 0$	**throw** *abort*
R	$hon = hon' = 0$	**throw** *abort*
F	$\emptyset$	**throw** *abort*

Figure 18: Package $\mathsf{Log}_n^{P,map}$

raw key.

Upon a query to $\mathsf{Q}_n(h)$, the oracle looks up the table Log_n and returns the handle h' if $\mathrm{Log}_n[h] = (h', \_, \_)$ (Read h is mapped to handle h').

Upon a query to $\mathsf{UNQ}_n(h, hon, k)$, the oracle first checks the mapping condition by searching the table for a handle h' such that (1) there exists an entry in the table with index h' ($\mathrm{Log}_n[h'] \neq \bot$), (2) h' maps to itself ($\mathrm{Log}_n[h'] = (h', hon', \_)$), (3) the entry contains the same key k ($\mathrm{Log}_n[h'] = (\_, hon', k)$, also read h' maps to the same key k), and (4) the mapping condition $map(\mathsf{level}(h), hon, \mathsf{level}(h'), hon', J_n[k])$ holds. If such an entry exists, a new Log entry (h', hon, k) is created with handle h as index (Read h is mapped to h') and the flag $J_n[k]$ is set to 1. The flag is used by mapping condition $map = 1$ which ensures the mapping happen only once for each key k. The found handle h' is finally returned from the oracle. This uniqueness functionality helps to detect key collisions and return the handle h' of an already existing key to the $\mathsf{SET}_{n,\ell}(h, hon, k)$ oracle of the key package. Recall that the key package returns the handle h' returned by $\mathsf{UNQ}_n(h, hon, k)$ if $h \neq h'$. Observe that this is always the case when a mapping occurs as the handle h does not exist in the table when the oracle $\mathsf{SET}_{n,\ell}(h, hon, k)$ does not return what $\mathsf{Q}_n(h)$ returns. If the mapping condition is 0, no table entry can satisfy the lookup and no mapping occurs. For ease of notation, we also remove the map parameter when $map = 0$. Notice that the existential quantifier can be seen as the epsilon operator of Hilbert epsilon calculus [AZ24].

If mapping is not triggered, the oracle $\mathsf{UNQ}_n(h, hon, k)$ checks the pattern condition by searching the table for a handle h^* such that (1) there exists an entry in the table with index h^* , (2) h^* maps to the same key k, and (3) the pattern condition $P_{cond}(\mathsf{level}(h), hon, \mathsf{level}(h'), hon')$ holds. If such an entry exists, the pattern command $P_{cmd}(\mathsf{level}(h), hon, \mathsf{level}(h'), hon')$ executes. Additionally, for the pattern $P = R$, if there exists an entry that maps to the key k but not one where the pattern condition $P_{cond}(\mathsf{level}(h), hon, \mathsf{level}(h'), hon')$ holds, then we abort with special symbol *win*. If the pattern condition is Z, no table entry can satisfy the search condition and the pattern command is not executed.

$\mathtt{Nkey}_n$

State

K_n : table

$\mathsf{SET}_n(h, hon, k)$

assert $\mathsf{name}(h) = n$
if $n = dh \wedge k = 0^{\mathsf{len}(k)}$:
  throw *abort*
if $\mathsf{Q}_n(h) \neq \bot$ **then**
  return $\mathsf{Q}_n(h)$
$h' \leftarrow \mathsf{UNQ}_n(h, hon, k)$
if $h' \neq h$ **then**
  return h'
$K_n[h] \leftarrow (k, hon)$
return h

$\mathsf{GET}_{n,\ell\in[d]}(h)$

if $n = 0salt$:
  assert $h = \mathsf{0salt}$
  return $(\mathsf{tag}_\bot(0), 0)$
if $n = 0ikm$:
  assert $h = \mathsf{0ikm}\langle alg\rangle$
  return $(\mathsf{tag}_{alg}(0^{\mathsf{len}(alg)}), 0)$
if $n = dh \wedge h = \mathsf{noDH}\langle alg\rangle$:
  return $(\mathsf{tag}_{alg}(0^{\mathsf{len}(alg)}), 0)$
assert $K_n[h] \neq \bot$
$(k^*, hon) \leftarrow K_n[h]$
$k \leftarrow \mathsf{tag}_h(k^*)$
return (k, hon)

Figure 17: Package $\mathtt{Nkey}_n$

Finally, if neither a mapping or pattern is triggered, a new entry (h, hon, k) in the table Log_n is created under the index h (i.e. h is mapped to itself with key k). Compared

to [BDLE+21], we have modified the code of this package to avoid notational mistakes and clarify the correct semantics for security analysis as well the as the verification.

We will use the following parameter combinations for the Log package in the security games and the security analysis: (O^* is the set of output keys)

$$\mathsf{Log}^{Z}_{dh}, \mathsf{Log}^{Z,\infty}_{dh}, \mathsf{Log}^{A}_{psk}, \mathsf{Log}^{A,1}_{psk}, \mathsf{Log}^{D,1}_{psk},$$
$$\mathsf{Log}^{Z}_{n\neq dh,psk}, \mathsf{Log}^{D}_{n\neq dh,psk,O^*}, \mathsf{Log}^{F}_{O^*}, \mathsf{Log}^{R}_{esalt}$$

Complementing our discussion on initialization of key table $K_{psk,0}$ in package $\mathsf{Key}_{psk,0}$, we mentioned that an initialization is required in the package $\mathsf{Log}^{P,map}_{psk}$. We initialize table Log_{psk} as follows: for $alg \in \mathcal{H}$, we set $\mathrm{Log}_{psk}[\mathsf{noPSK}\langle alg\rangle] = (\mathsf{noPSK}\langle alg\rangle, 0, 0^{\mathsf{len}(alg)})$. In the alternative solution to address the zero values for DH secrets, initialization of the table K_{dh} in package NKey_{dh} is necessary. Similarly, we initialize table Log_{dh} in package $\mathsf{Log}^{P,map}_{dh}$: for $alg \in \mathcal{H}$, $\mathrm{Log}_{dh}[\mathsf{noDH}\langle alg\rangle] = (\mathsf{noDH}\langle alg\rangle, 0, 0^{\mathsf{len}(alg)})$.

Observe that mapping conditions are only used for DH handles and PSK handles. Therefore, the Log table initialization are necessary to allow mapping conditions $map = \infty$ for DH handles and $map = 1$ for PSK handles to include noDH and noPSK handles. We will see how these initializations are taken into account in the verification.

Before defining the security games, we introduce the package Hash^b and pseudocode of functions $\mathsf{xtr}(k_1, k_2)$, $\mathsf{xpd}(k_1, (label, d))$, and $\mathsf{hash}(t)$. Package Hash^b is defined to be the package $\mathsf{Gacr}^{\mathsf{hash},b}$ where the function f is replaced with hash. Figure 19 shows the package $\mathsf{Gacr}^{\mathsf{f},b}$ and Figure 20 shows the pseudocode of $\mathsf{xtr}(k_1, k_2)$, $\mathsf{xpd}(k_1, (label, d))$, and $\mathsf{hash}(t)$, relying on functions $\mathsf{xpd}\text{-}alg(k_1, label, d)$ and $\mathsf{hmac}_{alg}(k_1, k_2)$ from the Figure 10. Notice the transcript t passed to the oracle $\mathsf{HASH}(t)$ is tagged with a hash algorithm (under adversarial control) and the corresponding algorithm $\mathsf{hash}\text{-}alg(t^*)$ is used. Therefore, the package is said to be agile as it handles several hash functions at the same time. BDEFKK use the pair of real ($\mathsf{Gacr}^{\mathsf{f},0}$) and ideal ($\mathsf{Gacr}^{\mathsf{f},1}$) games to define agile collision resistance security notions for functions $\mathsf{f} \in \{\mathsf{xpd}, \mathsf{xtr}, \mathsf{hash}\}$. See Lemma A.2 from [BDLE+21] where BDEFKK reduce agile collision resistance of these three functions to standard (non-agile) collision resistance assumptions for them.

Gacr^b

Parameters

b : bit

State

H : table

$\mathsf{HASH}(t)$

assert $\mathsf{alg}(t) \in \mathcal{H}$
if $H[t] \neq \perp$:
  return $H[t]$
$d \leftarrow \mathsf{untag}(\mathsf{f}(t))$
if $b \wedge d \in range(H)$:
  throw *abort*
$H[t] \leftarrow d$
return d

Figure 19: Package $\mathsf{Gacr}^{\mathsf{f},b}$

3.2.8 Security games

Having discussed all the necessary packages, we can formally define the real and ideal security games Gks^0 and $\mathsf{Gks}^1(\mathcal{S})$. Define $I^* := N \setminus (O^* \cup \{psk, dh\})$ to be the set of

$\mathsf{xtr}(k_1, k_2)$

if $\mathsf{alg}(k_1) = \perp$:
  $alg \leftarrow \mathsf{alg}(k_2)$
  $k_2 \leftarrow \mathsf{untag}(k_2)$
else $alg \leftarrow \mathsf{alg}(k_1)$
  $k_1 \leftarrow \mathsf{untag}(k_1)$
$k \leftarrow \mathsf{hmac}_{alg}(k_1, k_2)$
return $\mathsf{tag}_{alg}(k)$

$\mathsf{xpd}(k_1, (label, d))$

$alg \leftarrow \mathsf{alg}(k_1)$
$k_1 \leftarrow \mathsf{untag}(k_1)$
$k \leftarrow \mathsf{xpd}\text{-}alg(k_1, label, d)$
return $\mathsf{tag}_{alg}(k)$

$\mathsf{hash}(t)$

$t^* \leftarrow \mathsf{untag}(t)$
$alg \leftarrow \mathsf{alg}(t)$
$d^* \leftarrow \mathsf{hash}\text{-}alg(t^*)$
$d \leftarrow \mathsf{tag}_{alg}(d^*)$
return d

Figure 20: Pseudocode of $\mathsf{xtr}(k_1, k_2)$, $\mathsf{xpd}(k_1, (label, d))$, and $\mathsf{hash}(t)$

all internal keys. Let $\Pi_{i\in I} M_i$ denote the parallel composition of packages M_i for all indices $i \in I$ and index set I. For any package $\mathcal{S}$ with the following input and output interface, we define

$$\mathtt{Gks}^0 := \left(\begin{array}{c} \mathtt{DH} \\ \hline \mathtt{ID}_{\mathsf{SET}_{psk,0}} \\ \hline \Pi_{n\in N_{\mathsf{xtr}},\ell\in[d]}\mathtt{Xtr}^0_{n,\ell} \\ \hline \mathtt{Check} \circ \dfrac{\Pi_{n\in N_{\mathsf{xpd}},\ell\in[d]}\mathtt{Xpd}_{n,\ell}}{\Pi_{\ell\in[d]}\mathtt{ID}_{\mathsf{GET}_{binder,\ell}}} \\ \hline \Pi_{n\in O^*,\ell\in[d]}\mathtt{ID}_{\mathsf{GET}_{n,\ell}} \end{array}\right) \circ \left(\begin{array}{c} \mathtt{NKey}_{dh} \circ \mathtt{Log}^Z_{dh} \\ \hline \left(\mathtt{Key}^1_{psk,0} | \Pi_{\ell\in\{1,\ldots,d+1\}}\mathtt{Key}^0_{psk,\ell}\right) \circ \mathtt{Log}^A_{psk} \\ \hline \mathtt{Hash}^0 \\ \hline \mathtt{NKey}^0_{0salt} \\ \hline \mathtt{NKey}^0_{0ikm} \\ \hline \Pi_{n\in I^*}\left((\Pi_{\ell\in[d]}\mathtt{Key}^0_{n,\ell}) \circ \mathtt{Log}^Z_n\right) \\ \hline \Pi_{n\in O^*}\left((\Pi_{\ell\in[d]}\mathtt{Key}^0_{n,\ell}) \circ \mathtt{Log}^Z_n\right) \end{array}\right)$$

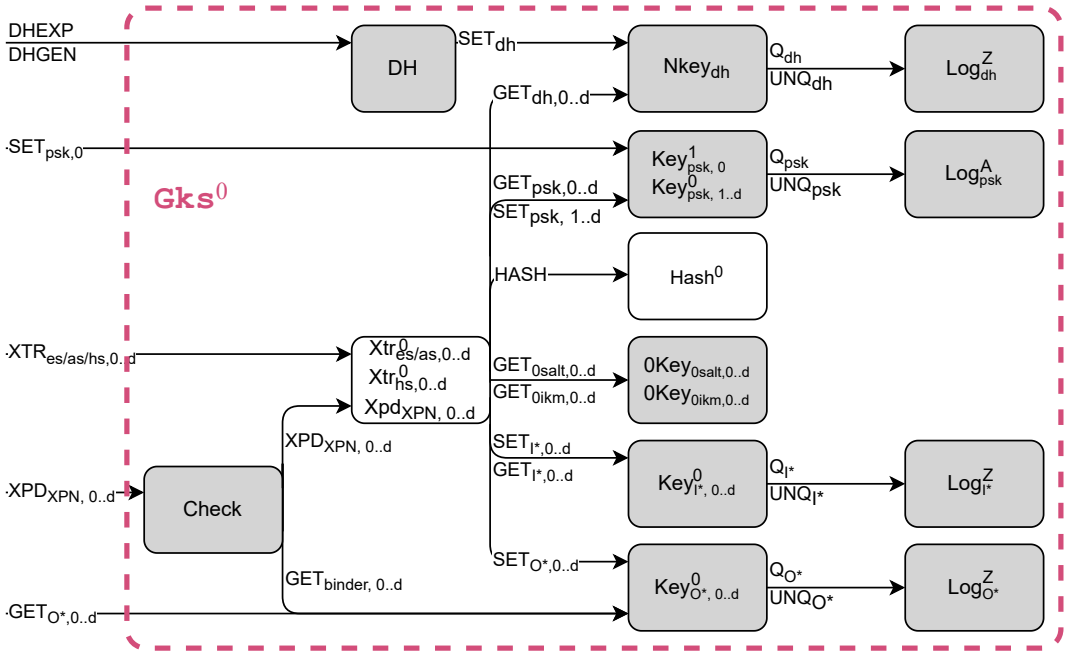
$$\mathtt{Gks}^1(\mathcal{S}) := \frac{\mathcal{S}}{\Pi_{n\in O^*,\ell\in[d]}\mathtt{ID}_{\mathsf{GET}_{n,\ell}}} \circ \Pi_{n\in O^*}\left((\Pi_{\ell\in[d]}\mathtt{Key}^1_{n,\ell}) \circ \mathtt{Log}^F_n\right).$$

The following tables summarize the input and output interfaces of packages used in the games:

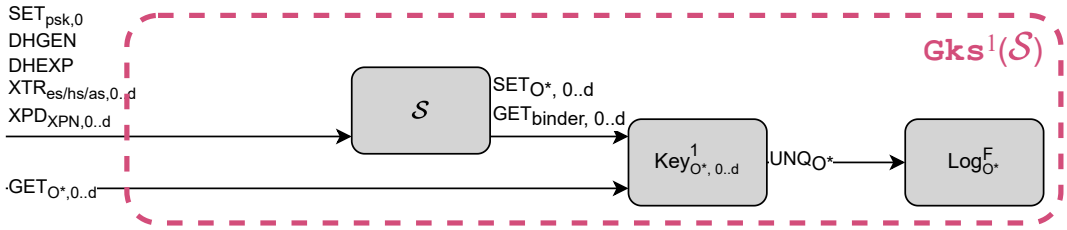
Package	Input Interface
DH	SET_{dh}
Check	$\mathsf{XPD}_{n,\ell}$ for $n \in N_{\mathsf{xpd}}, \ell \in [d]$
$\mathtt{Key}^b_{n,\ell}$ for $n \in N \setminus \{dh, 0salt, 0ikm\}, \ell \in [d]$	$\mathsf{Q}_n, \mathsf{UNQ}_n$
$\mathtt{Log}^{P,map}_n$ for $n \in N \setminus \{0salt, 0ikm\}$	$\varnothing$
$\mathtt{NKey}_{dh}$	$\mathsf{Q}_{dh}, \mathsf{UNQ}_{dh}$
$\mathtt{NKey}_n$ for $n \in \{0salt, 0ikm\}$	$\varnothing$
$\mathtt{Xtr}^b_{n,\ell}$ for $n \in N_{\mathsf{xtr}}, \ell \in [d]$	$\mathsf{GET}_{n_1,\ell}, \mathsf{GET}_{n_2,\ell}, \mathsf{SET}_{n,\ell}$ where $(n_1, n_2) = \mathsf{PrntN}(n)$
$\mathtt{Xpd}_{n,\ell}$ for $n \in N_{\mathsf{xpd}} \setminus \{psk\}, \ell \in [d]$	$\mathsf{GET}_{n_1,\ell}, \mathsf{SET}_{n,\ell}$ where $(n_1, \_) = \mathsf{PrntN}(n)$
$\mathtt{Xpd}_{psk,\ell}$ for $\ell \in [d]$	$\mathsf{GET}_{n_1,\ell}, \mathsf{SET}_{n,\ell+1}$ where $(n_1, \_) = \mathsf{PrntN}(n)$
$\mathcal{S}$	$\mathsf{GET}_{binder,\ell}, \mathsf{SET}_{n,\ell}$ for all $n \in O^*, \ell \in [d]$

Package	Output Interface
DH	DHEXP, DHGEN
Check	$\mathsf{XPD}_{n,\ell}$ for $n \in N_{\mathsf{xpd}}, \ell \in [d]$
$\mathtt{Key}^b_{n,\ell}$ for $n \in N \setminus \{dh, 0salt, 0ikm\}, \ell \in [d]$	$\mathsf{SET}_{n,\ell}, \mathsf{GET}_{n,\ell}$
$\mathtt{Log}^{P,map}_n$ for $n \in N \setminus \{0salt, 0ikm\}$	$\mathsf{Q}_n, \mathsf{UNQ}_n$
$\mathtt{NKey}_{dh}$	$\mathsf{SET}_{dh}, \mathsf{GET}_{dh,\ell}$ for all $\ell \in [d]$
$\mathtt{NKey}_n$ for $n \in \{0salt, 0ikm\}$	$\mathsf{GET}_{n,\ell}$ for all $\ell \in [d]$
$\mathtt{Xtr}^b_{n,\ell}$ for $n \in N_{\mathsf{xtr}}, \ell \in [d]$	$\mathsf{XTR}_{n,\ell}$
$\mathtt{Xpd}_{n,\ell}$ for $n \in N_{\mathsf{xpd}} \setminus \{psk\}, \ell \in [d]$	$\mathsf{XPD}_{n,\ell}$
$\mathtt{Xpd}_{psk,\ell}$ for $\ell \in [d]$	$\mathsf{XPD}_{n,\ell}$
$\mathcal{S}$	$\mathsf{DHEXP}, \mathsf{DHGEN}, \mathsf{XTR}_{N_{\mathsf{xtr}},[d]}, \mathsf{XPD}_{N_{\mathsf{xpd}},[d]}, \mathsf{SET}_{psk,0}$

Figure 21 shows a compact visualization of the games; the oracles exposed to the adversary are marked on the arrows at the left of the diagrams. We want to emphasize



(a) Real game $\texttt{Gks}^0$



(b) Ideal game $\texttt{Gks}^1(\mathcal{S})$

Figure 21: Key schedule security games for d levels with output keys O^*, internal keys $I^* := N \setminus (O^* \cup \{psk, dh\})$, $XPN := N_{\mathsf{xpd}}$, $\texttt{0Key}_{0salt} := \texttt{NKey}_{0salt}$ and $\texttt{0Key}_{0ikm} := \texttt{NKey}_{0ikm}$ (copied with permission from [BDLE+21])

that an extended visualization look like a long *ladder* with zigzag steps. Keying and keyed packages `Xtr` and `Xpd` lie on the left *rail* while the key and log packages `Key`, `NKey`, and `Log` lie on the right rail such that any keying package SETs a key in a key package `Key` from which another keyed package GETs a key. Package $\texttt{Key}_{psk,0}$ lies on the top step and the package $\texttt{Key}_{es,0}$ sits on the step below it and all other packages exist sit them in the order of computation in the key schedule. As a result, the number of *steps* of the ladder grows with the number of resumption levels d.

Notice that in the game $\texttt{Gks}^0$, the only idealized key package is $\texttt{Key}^1_{psk,0}$. The adversary can generate honest application PSKs by calling the oracle $\mathsf{SET}_{psk,0}(h, hon, k)$ with $hon = 1$ and dishonest application PSKs by setting $hon = 0$. Recall the security goals of the key schedule: honest output keys O^* shall be pseudorandom and unique. Using the handles, we translate these two properties as follows: (pseudorandom) for any honest output key handle h (i.e. $\mathsf{name}(h) \in O^*$), the key returned by the oracle $\mathsf{GET}_{\mathsf{name}(h),\mathsf{level}(h)}(h)$ can not be distinguished from a uniformly random key and

(uniqueness) for any two distinct output key handles $h \neq h'$ with the same name (i.e. $\mathsf{name}(h) = \mathsf{name}(h') = n \in O^*$), the keys returned by the queries $\mathsf{GET}_{n,\mathsf{level}(h)}(h)$ and $\mathsf{GET}_{n,\mathsf{level}(h')}(h')$ are different. Remember that a handle is honest if an honest DH handle or an honest PSK handle is used at some point during its construction. This was ensured by the oracle XTR that computed the honesty bit of a new handle by taking a logical OR of the honesty bits of its parents and the oracle XPD that set the honesty bit of a new handle the same as its parent.

These two handle-based definitions can now be easily implemented in the game $\mathtt{Gks}^1(\mathcal{S})$ as the ideal functionality $\Pi_{n\in O^*}\left((\Pi_{\ell\in[d]}\mathtt{Key}^1_{n,\ell}) \circ \mathtt{Log}^F_n\right)$. The idealized output key packages $\mathtt{Key}^1_{n,\ell}$ store random keys for honest handles, achieving pseudorandomness. The F pattern of the package $\mathtt{Log}^F_n$ ensures full uniqueness functionality such that it aborts upon the first output key collision, achieving uniqueness. Notice that if two TLS sessions use the same external/application PSK shared out-of-band, the key schedule of their sessions might derive the same set of keys. Although this can only happen if the sessions use the same set of nonces and DH secrets (and other arguments) but recall that such arguments are under adversarial control in our model. Therefore, the adversary can set the same dishonest application PSKs under two different handles and derive distinct output key handles with the same keys. This violates the uniqueness property. To prevent this trivial attack, we use the A pattern for the package $\mathtt{Log}^A_{psk}$ in the game $\mathtt{Gks}^0$. The A pattern implements application PSK uniqueness functionality. Namely, the oracle UNQ does not allow any collision between dishonest application PSKs chosen by the adversary. (Figure 18 defines the pattern condition $P_{cond}(r, hon, r', hon') := hon = hon' = 0 \wedge r = r' = 0$ when $P = A$.)

3.3 Overview of Key Schedule Security Analysis

BDDEFF construct a concrete and efficient simulator package $\mathcal{S}$ and in Theorem C.1 of [BDLE+21] bound the advantage $\mathsf{Adv}(\mathcal{A}, \mathtt{Gks}^0, \mathtt{Gks}^1(\mathcal{S}))$ of any adversary $\mathcal{A}$, which makes queries for at most d resumption levels, by the advantage of adversaries of the form $\mathcal{A} \rightarrow \mathcal{R}$, where $\mathcal{R}$ is a PPT reduction, against the modular security games for (dual) pseudorandomness and collision resistance assumption of xtr and xpd functions, collision resistance assumption of hash function hash, and Salted Oracle Diffie-Hellman (SODH) assumption.

These modular security games for the assumption are specific compositions of the packages already illustrated in the previous section. In Appendix E of [BDLE+21], BDEFKK bound the advantage of any adversary $\mathcal{A}$ against these modular security games by the advantage of adversaries of the form $\mathcal{A} \rightarrow \mathcal{R}$ for PPT reductions $\mathcal{R}$ against standard monolithic security games for the aforementioned assumptions. For example, see the monolithic security games for SODH assumption in Section 5. Therefore, they reduce the security of key schedule to standard assumptions in the concrete security setting. (See Section 2.1 for the discussion on concrete security model.)

We refer the reader to [BDLE+21] for the detailed security reduction of modular

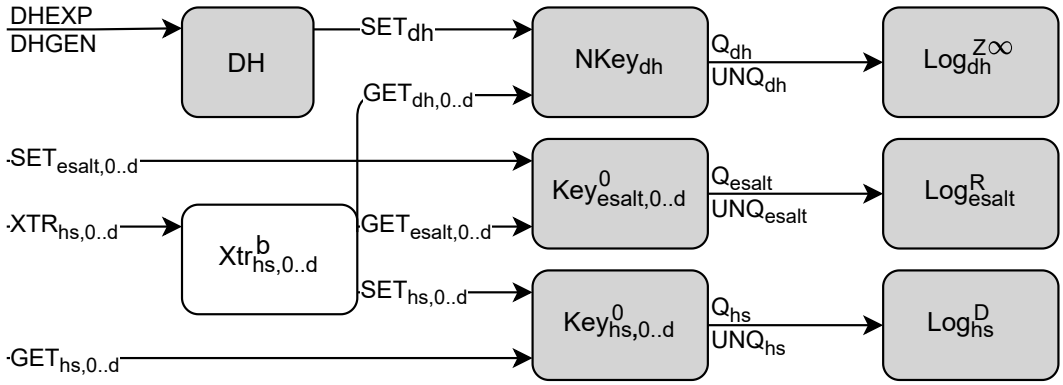


Figure 22: Games $\texttt{Gsodh}^b$ for $b \in \{0, 1\}$ (copied with permission from [BDLE+21])

to standard monolithic assumptions. However, we briefly sketch an overview of the reduction of key schedule security to modular assumptions (i.e. proof of Theorem C.1). Towards this end, we illustrate three modular assumptions to motivate for the main lemmata of the proof. To allow the reader to compare an example of modular assumption with a monolithic assumption, we begin with the modular SODH assumption.

3.3.1 Modular SODH assumption

Figure 22 shows the definition of real ($b = 0$) and ideal ($b = 1$) modular SODH security games $\texttt{Gsodh}^b$ as the compositions of packages illustrated in the previous section. The modular SODH assumption states that the games $\texttt{Gsodh}^b$ for $b \in \{0, 1\}$ are indistinguishable for any adversary $\mathcal{A}$ against them. In Lemma E.1 of [BDLE+21], BDEFKK reduce the modular SODH assumption to collision resistance of xtr and the monolithic SODH assumption (introduced in Section 5). Notice that an adversary against $\texttt{Gsodh}^b$ can set arbitrary early salts k_{esalt} in the key package $\texttt{Key}^0_{esalt,[d]}$ under honest and dishonest handles since the key package is not idealized and a random key is not sampled for the honest handles. Informally, the assumption states that $\mathsf{xtr}(k_{dh}, k_{esalt})$ is pseudorandom for honest DH secret k_{dh} and adversarially chosen k_{esalt} (cf. Oracle Diffie-Hellman assumption [ABR01] explained in Section 5).

If the adversary is allowed to set the same early salt key k_{esalt} under two distinct handles $h \neq h'$ in level ℓ and mix this key with an honest DH secret under the handle h_{dh}, $\texttt{Xtr}^1_{hs,\ell}$ (in the ideal game) samples two distinct keys for h_{esalt} and h'_{esalt} with high probability and set them in the key packge $\texttt{Key}^0_{hs,\ell}$. The adversary can easily distinguish the real and ideal games $\texttt{Gsodh}^b$ by first setting $h_{hs} = \mathsf{XTR}_{hs,\ell}(h_{esalt}, h_{dh})$ calling $h_{hs} = \mathsf{XTR}_{hs,\ell}(h_{esalt}, h_{dh})$ and $h'_{hs} = \mathsf{XTR}_{hs,\ell}(h'_{esalt}, h_{dh})$ and comparing there outputs of $\mathsf{GET}_{hs,\ell}(h_{hs})$ and $\mathsf{GET}_{hs,\ell}(h'_{hs})$. They are the the same in the real game $\texttt{Gsodh}^0$ and distinct with high probability in the ideal game $\texttt{Gsodh}^1$. To prevent this trivial attack and similar to the choice of A pattern for the package $\texttt{Log}^A_{psk}$ in $\texttt{Gks}^0$, the pattern R is used for package $\texttt{Log}^R_{esalt}$ to abort whenever the adversary sets the same early salt under two distinct handles. Recall that the pattern R aborts with the symbol

abort when two dishonest handles have a key collision (regardless of the levels of the handles) and with the symbol *win* when an honest handle has the same key as another handle (again regardless of the levels). The reason for aborting with a different symbol *win* when an honest handle has the same key as a dishonest handle is to reduce to pseudorandomness of xpd operations deriving k_{esalt}. Intuitively, honest handles map to random keys sampled by an idealized key package. BDEFKK bound the probability that such keys collide with adversarially chosen keys or keys derived from dishonest base keys using the pseudorandomness assumptions for xpd function.

3.3.2 Core key schedule security: Hybrid argument

BDEFKK reduce to the modular SODH assumption as a game hop in their hybrid argument and idealize the package $\mathtt{Xtr}^{b}_{hs,[d]}$. Generally speaking, when proving composition $f(g(k_2, y), x)$ of pseudorandom functions $f(k_1, x)$ and $g(k_2, y)$ is a pseudorandom function, we first reduce to psuedorandomness assumption of g and replace $g(k_2, y)$ with a random key k_1 and then reduce to psuedorandomness assumption of f and replace $f(k_1, x)$ with a random string r. BDEFKK consider the key schedule as a complex chain of psuedorandomness extractors and functions. They use a similar approach by idealizing the xtr and xpd operations layer by layer (i.e. one operation (key) in each resumption level and then one level at a time). However, They idealize all early salt extractors at once with the reduction to modular SODH assumption because of the global key package $\mathtt{NKey}_{dh}$ and the lack of an idealziation parameter for this package. [15] Nevertheless, they idealize all other key packages one package at a time by reducing to the corresponding pseudorandomness assumption for the xpd operation. [16] (They go down the *ladder* visualization of the game $\mathtt{Gks}^0$ one *step* at a time.)

For instance, in a similar setting, for any $\ell \in [d]$, Figure 23 shows the definition of real ($b = 0$) and ideal ($b = 1$) security games $\mathtt{Gxtr2}^{b}_{hs,\ell}$ for modular pseudorandomness assumption of $k_{hs} = \mathsf{xtr}(k_{esalt}, k_{dh})$ operation. The assumption states that the games $\mathtt{Gxtr2}^{b}_{hs,\ell}$ for $b \in \{0, 1\}$ are indistinguishable for any adversary $\mathcal{A}$ against them. Informally, it states that k_{hs} is psuedorandom if k_{esalt} is chosen randomly. Notice that honest early salt k_{esalt} is chosen randomly by the key package $\mathtt{Key}^{1}_{esalt,\ell}$ and k_{dh} is directly chosen and set by the adversary (not necessarily a group element) with arbitrary honesty status. Observe that the key k_{hs} is stored in the key package $\mathtt{Key}^{b}_{hs,\ell}$ which stores a random key for honest handle in the ideal game $b = 1$ and the actual key in the real game $b = 0$. Other modular pseudorandomness assumption games have a similar structure where the input key packages are idealized and the output key packages (output of xtr or xpd, not to be confused with the output keys of key schedule) inherit the idealization bit b of the game. As a result, the input keys for the

[15] This is indeed the essential reason for existence of Oracle Diffie-Hellman assumption introduced by Abdalla, Bellare, and Rogaway . Some favor of the assumption is crucial for security analysis of Diffie-Hellman-based protocols where the DH secret is hashed with hash function H or fed into a key derivation function such as xtr (i.e. hmac). Since DH secrets g^{xy} can not be idealized, the assumption idealizes the hash value $H(g^{xy})$ of the secrets.

[16] $\mathtt{Xtr}^{b}_{hs,[d]}$ is the only Xtr package that is idealized.

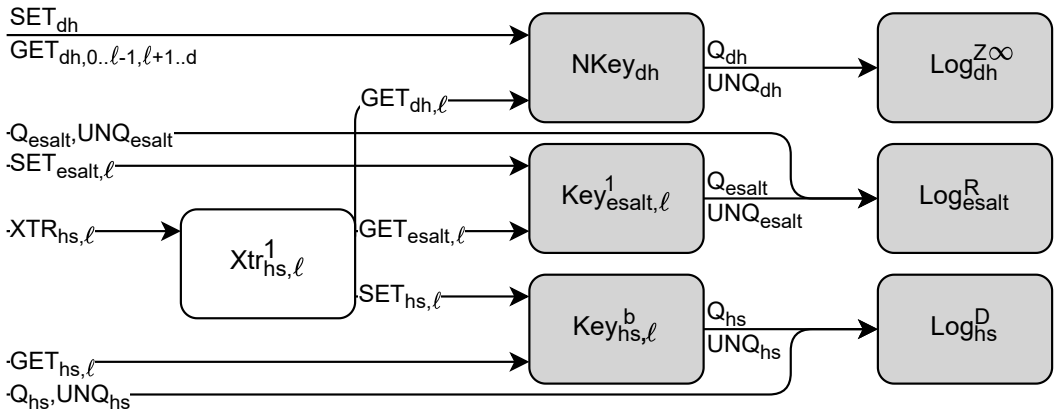


Figure 23: Games $\texttt{Gxtr2}^{b}_{hs,\ell}$ for $b \in \{0,1\}$ (copied with permission from [BDLE+21])

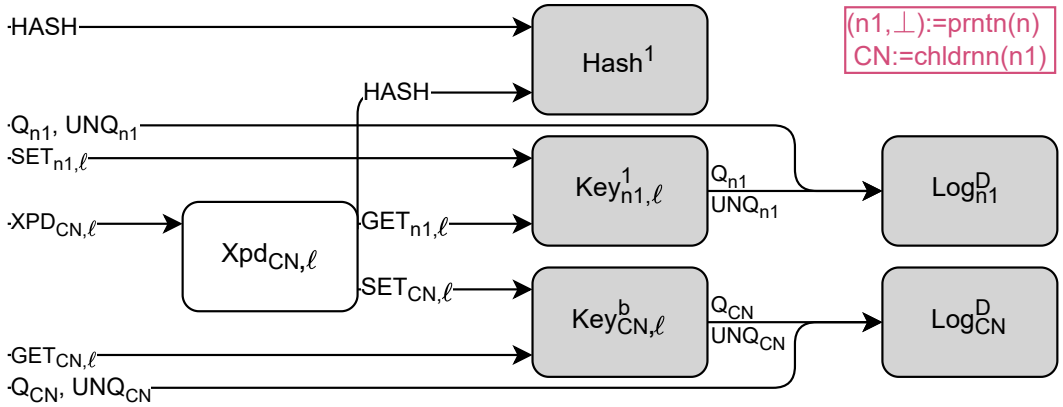


Figure 24: Games $\texttt{Gxpd}^{b}_{n,\ell}$ for $b \in \{0,1\}$ (copied with permission from [BDLE+21])

next xtr or xpd operation (children of a node in the parenting graph 11) in the key schedule are stored in an idealized key package, which allows to reduce the modular psuedorandomness assumption for the next operation. Naturally, the hybrid argument begins with the idealized key package $\texttt{Key}^{1}_{psk,0}$ for the application PSKs. Figure 24 shows the security games of another modular pseudorandomness assumption for xpd operations but with the similar pattern for the key packages. The game is parameterized by an expand key name n (with parent n_1) and level ℓ. The input key package $\texttt{Key}^{1}_{n_1,\ell}$ is idealized by the previous reduction and the output key packages $\texttt{Key}^{b}_{CN,\ell}$ inherit the idealization parameter from the game. Notice that all the keys CN derived from the same parent key n_1 are idealized at the same time.

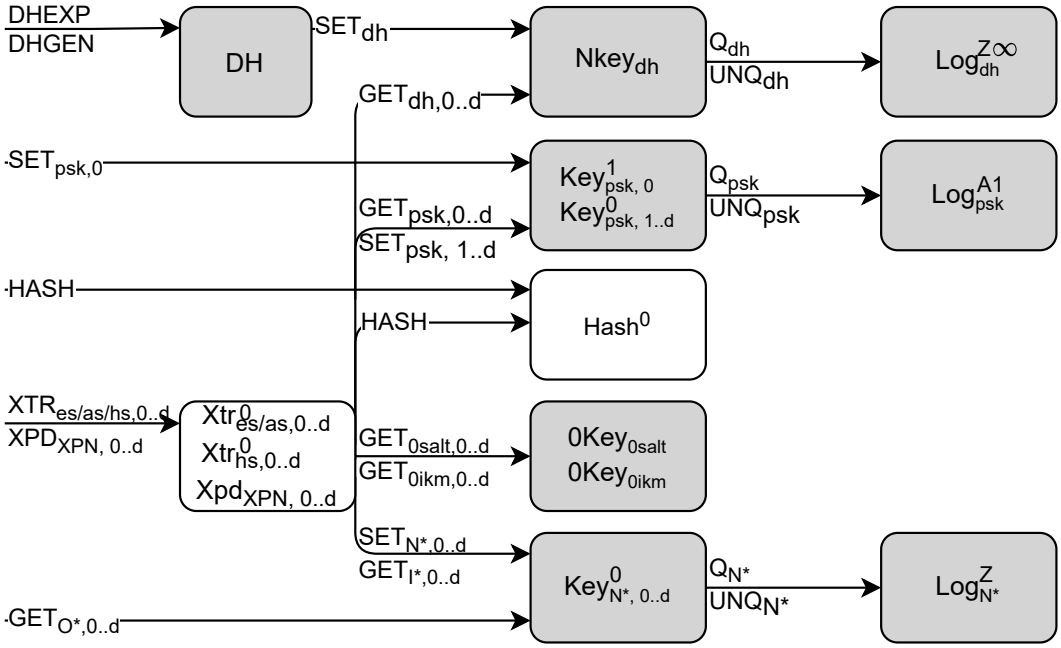
At the end of the hybrid argument, BDEFKK arrive at the game $\texttt{Gks}^{1}(\mathcal{S})$ where $\mathcal{S}$ is the composition of some of the existing packages and, more importantly, output key packages $\texttt{Key}^{1}_{O^*,[d]}$ [17] are idealized. Moreover, these idealized key packages are backed by the Log packages $\texttt{Log}^{F}_{O^*}$ with full uniqueness functionality. Similar to pattern R, the pattern F also does not tolerate any key collisions but always aborts with the same symbol $abort$ when two handles have the same keys, regardless of the levels and honesty bits. Since the key package $\texttt{Key}^{1}_{O^*,[d]}$ is idealized, honest keys are sampled randomly and, intuitively, the collision probability of honest keys can be bounded with a birthday bound while causing collisions of honest and dishonest keys should be reduced to computational assumptions for xpd operations. Concretely, in two game hops, BDEFKK replace the packages $\texttt{Log}^{D}_{O^*}$ with $\texttt{Log}^{F}_{O^*}$ as well as the packages $\texttt{Log}^{R}_{esalt}$ with $\texttt{Log}^{D}_{esalt}$ by reducing to modular pre-image resistance assumptions for xpd functions, which in turn they reduce to standard pseudorandomness of xpd. Recall that the D pattern only aborts when two dishonest handles have a key collidion. As a result, it suffices to show that at the end of the hybrid argument we arrive at a game with Log packages $\texttt{Log}^{D}_{O^*}$. From here, we make two additional game hops as described to arrive at the game with packages $\texttt{Log}^{F}_{O^*}$. Interestingly, the pattern D (dishonest key uniqueness) propapagtes backwards in the key schedule due to the collision resistance of the xpd and xtr operations. Since handles resemble key derivations steps, collision

[17] $\texttt{Key}^{1}_{O^*,[d]}$ is an abuse of notation when we refer to the collection of packages $\{\texttt{Key}^{1}_{n,\ell}\}_{n\in O^*,\ell\in[d]}$.

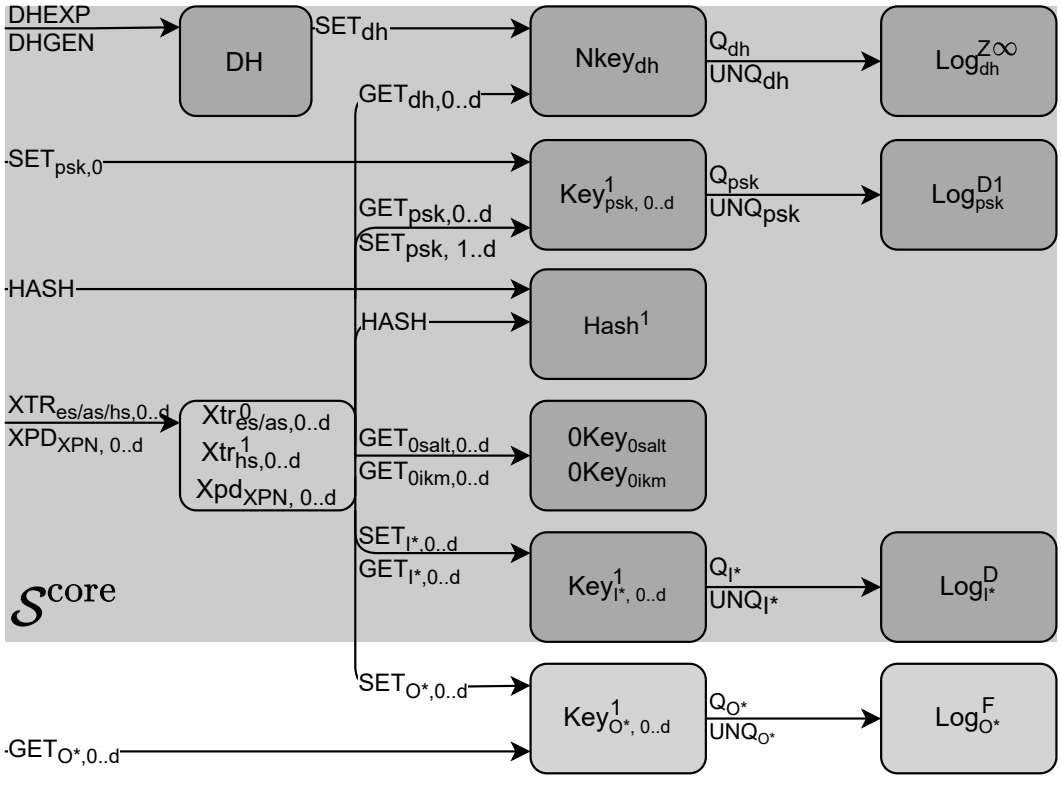
resistance of the key derivation functions translate to the difficulty of constructing two distinct extract or expand handles using the XPD and XTR oracles that map to the same key. For example, if the key derived by $\mathsf{XPD}(h_1, r, args)$ and $\mathsf{XPD}(h_2, r, args)$ are the same, after reducing to collision resistance assumption of xpd and xtr, we can conclude h_1 and h_2 map to the same key in the corresponding Log packages. The D pattern back propagation (up the *ladder* of game $\mathtt{Gks}^0$) transitively brings dishonest key uniqueness assertions closer to the base keys: DH secrets and PSKs. Therefore, assuming collision resistance of xtr and xpd operations, uniqueness of dishonest base keys transitively imply uniqueness of output keys. In Lemma D.3 of [BDLE+21], BDEFKK indeed prove this claim by replacing all Z and A patterns (except for $\mathtt{Log}^Z_{dh}$) of the Log packages with D pattern. Lemma D.3 is itself a mini hybrid argument with code equivelances and reductions to the collision resistance assumptions of xpd and xtr operations. In an important code equivalence proved in Claim D.7.1 of [BDLE+21], BDEFKK show that all Z and A patterns can be replaced with D patterns, after having made xtr and xpd operations collision-free. As we will see, uniqueness of dishonest base keys are guaranteed by the mapping parameters $map = \infty$ and $map = 1$ of the Log packages $\mathtt{Log}^{Z\infty}_{dh}$ and $\mathtt{Log}^{A1}_{psk}$.

To summarize the hybrid argument, we present the initial (H_0) and final (H_{8d+6}) games of the hybrid argument, where d is the number of resumption levels. Figures 25a and 25b depict the visualization of the games $\mathtt{Gcore}^0$ and $\mathtt{Gcore}^1$, respectively. Again, we consider the figures as the definition of the games. BDEFKK refer to the games $\mathtt{Gcore}^0$ and $\mathtt{Gcore}^1$ by the core key schedule security games and prove in the core key schedule theorem via the aforementioned idealization hybrid argument that $\mathtt{Gcore}^0$ and $\mathtt{Gcore}^1$ are indistinguishable. Concretely, they define $H_0 := \mathtt{Gcore}^0$ after $8d + 6$ game hops, arrive at $H_{8d+6} := \mathtt{Gcore}^1$. Each game hop is an SSP-style reduction (graph cut) to the modular computational assumptions. One of the game hops reduce to the modular SODH assumption. Another game hop repalces A and Z patterns with D pattenrs by reducing to the collision resistance of xtr and xpd operations. One game hop reduces to the collision resistance of hash function hash used for computing hash digest of transcripts. One game hop replaces the D pattern of output key packages with the F pattern by reducing to modular pre-image resistance assumption. In two other game hops, the pattern D of the package $\mathtt{Key}^0_{esalt,[d]}$ is replaced with the pattern R (for reduction to SODH assumption) and then replaced back with the pattern D. The other $8d$ game hops idealize the key packages in each of the d levels. One can define an equivalence class on the TLS 1.3 key names such that two keys are equivalent if they are derived from the same parent key. For example, client application traffic k_{sat} and server application traffic k_{sat} are equivalent because both are derived with an expand operation applied the same handshake secret k_{hs} but different labels. This equivalence relation induces eight classes: three singleton classes for the extract keys ($\{es\}$, $\{hs\}$, $\{as\}$) and five classes for the expand keys ($\{eem, cet, bind, esalt\}$, $\{cht, sht, hsalt\}$, $\{sat, cat, rm\}$, $\{psk\}$) The idealization order in each level idealizes one equivalence class at a time, i.e. all the children of the same parents at the same time. Hence, $8d$ game hops are necessary for idealization of all key packages.

We refer the reader to Appendix A of [BDLE+21] for the definition of all modular



(a) Game $\texttt{Gcore}^0$



(b) Game $\texttt{Gcore}^1$

Figure 25: Core key schedule security games $\texttt{Gcore}^b$ (copied with permission from [BDLE+21]): observe that package `Check` is not needed for idealization

assumptions and Appendix D for the formal definition of the hybrid games and reductions to modular assumptions.

3.3.3 Mapping parameters

We now explain why the infinity mapping pattern $map = \infty$ is used for the package $\mathtt{Log}^{Z\infty}_{dh}$ and dishonest key uniqueness functionality is propapagted from the $\mathtt{Log}^{D}_{O^*}$ to $\mathtt{Log}^{Z\infty}_{dh}$. Imagine in the game $\mathtt{Gxtr2}^{b}_{hs,\ell}$, the adversary is allowed to set the same DH secret (say $k_{dh} = g^{xyr}$) under two distinct dishonest DH handles (say $h_{dh} = \mathsf{dh}\langle\mathsf{sort}(g^x, (g^y)^r)\rangle$ and $h'_{dh} = \mathsf{dh}\langle\mathsf{sort}(g^y, (g^x)^r)\rangle$ for honest shares g^x and g^y but adversarially chosen r) and mixes them with an honest handle h_{esalt}. The adversary can set these handle by calling $h_{dh} = \mathsf{DHEXP}(g^x, (g^y)^r)$ and $h'_{dh} = \mathsf{DHEXP}(g^y, (g^x)^r)$. Two distinct dishonest handles $h_{hs} \neq h'_{hs}$ are returned by $\mathsf{XTR}_{hs,\ell}$ while two distinct (with high probability) keys $k_{hs} \neq k'_{hs}$ are sampled by the oracles $\mathsf{SET}_{hs,\ell}(h_{hs}, 1, \mathsf{xtr}(k_{esalt}, g^{xyr}))$ and $\mathsf{SET}_{hs,\ell}(h'_{hs}, 1, \mathsf{xtr}(k_{esalt}, g^{xyr}))$ of the idelaized key package $\mathtt{Key}^{1}_{hs,\ell}$ in the ideal game $\mathtt{Gxtr2}^{1}_{hs,\ell}$. An adversary can easily distinguish the real and ideal games by comparing the output keys from $\mathsf{GET}_{hs,\ell}(h_{hs})$ and $\mathsf{GET}_{hs,\ell}(h'_{hs})$. These two keys are the same in the real game but distinct with high probability in the ideal game. This trivial attack is prevented with the infinity mapping pattern $map = \infty$. (It could have also been prevented by $\mathtt{Log}^{D}_{dh}$.) Instead of *directly* aborting the game (imagine $\mathtt{Log}^{D}_{dh}$) when two distinct dishonest DH handles correspond to the same DH secret value, BDEFKK use the mapping $map = \infty$ parameter for the package $\mathtt{Log}^{Z\infty}_{dh}$ to *indirectly* abort the game in case of such a collision. (They delay the game abort.) We illustrate the infinity mapping behaviour of the package with an example. Let $h_1 := h_{dh}$ be the previously defined dishonest DH handle for the previously defined DH secret $k_1 := k_{dh}$ set via oracle SET_{dh} of $\mathtt{NKey}_{dh}$ such that $K_{dh}[h_1] = (k_1, 0)$ and $\mathrm{Log}_{dh}[h_1] = (h_1, 0, k_1)$. When $\mathsf{SET}_{dh}(h_2, 0, k_1)$ is queried for the DH handle $h_2 := h'_{dh}$, $\mathsf{UNQ}_{dh}(h_2, 0, k_1)$ is queried to the Log package and $\mathrm{Log}_{dh}[h_2] = (h_1, 0, k_1)$ is set (h_2 is mapped to h_1) and h_1 is returned. Since $h_2 \neq h_1$, h_1 is also returned from $\mathsf{SET}_{dh}(h_2, 0, k_1)$ and $K_{dh}[h_2]$ is not set. As a result, the adversary is forced to only use the handle h_1 to refer to the key k_1 and if it queries $\mathsf{XTR}_{hs,\ell}(h_{esalt}, h_2)$ with handle h_2, the oracle $\mathsf{GET}_{dh,\ell}(h_2)$ aborts (indirect abort) when asserting $K_{dh}[h_2] \neq \perp$. However, the game does not abort (delayed abort) if the adversary refrains from such a query on non-existing entries. The adversary notices a mapping has occured when it receives h_1 instead of the expected handle h_2 from the second query $\mathsf{DHEXP}(g^y, (g^x)^r)$ which calls $\mathsf{SET}_{dh}(h_2, 0, k_1)$.

As motivated in Section 3.1.1, an adversary can easily force two TLS sessions to share the same DH secret. (A man-in-the-middle attacker can choose r and replace clieant and server shares with $(g^x)^r$ and $(g^y)^r$, respectively.) BDEFKK choose not to abort the games on such easily created collisions immediately, but rather delay the abort to future if the adversary is curious to query the key package on non existing entry. This indirect or delayed abort behaviour aligns with the mapping behaviour of the Log package. As a result, the idealization hybrid argument requires the package $\mathtt{Log}^{Z\infty}_{dh}$ with infinity mapping to reduce to the security games $\mathtt{Gsodh}^{b}$ and $\mathtt{Gxtr2}^{b}$. However, the game $\mathtt{Gks}^{0}$ uses the package $\mathtt{Log}^{Z}_{dh}$ without the mapping. BDEFKK prepares $\mathtt{Gks}^{0}$

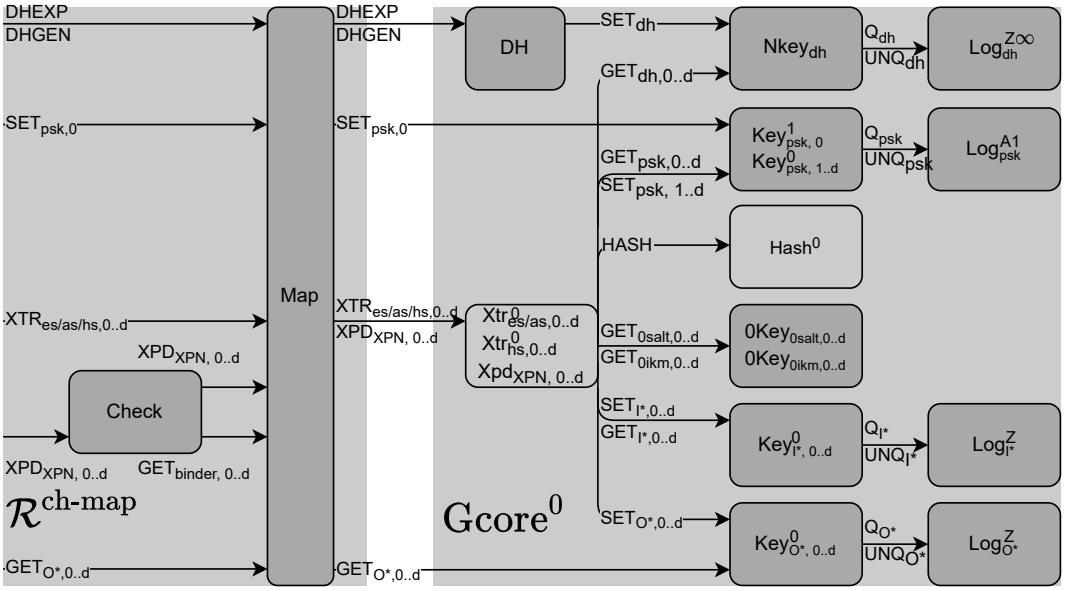


Figure 26: Game $\texttt{Gks}^{\texttt{0Map}}$ (copied with permission from [BDLE+21])

for the indealization hybrid argument with a game hop in Lemma C.2 of [BDLE+21] by replacing $\texttt{Gks}^{0}$ with the code equivalent game $\texttt{Gks}^{\texttt{0Map}}$. The game $\texttt{Gks}^{\texttt{0Map}}$ replaces the package $\texttt{Log}^{Z}_{dh}$ with $\texttt{Log}^{Z\infty}_{dh}$ at the cost of introducing a new mapping package `Map`. Figure 26 shows visualization of the game $\texttt{Gks}^{\texttt{0Map}}$. We consider the figure as the formal game definition and refrain from defining it with parallel and sequential composition (as we did for $\texttt{Gks}^{0}$ and $\texttt{Gks}^{1}(\mathcal{S})$). The only differences between $\texttt{Gks}^{0}$ and $\texttt{Gks}^{\texttt{0Map}}$ are the packages $\texttt{Log}^{Z\infty}_{dh}$, $\texttt{Log}^{A1}_{psk}$, and `Map`. We will shortly discuss the mapping $map = 1$ in the package $\texttt{Log}^{A1}_{psk}$.

The idea of introducing the package `Map` is to proxy all oracle calls and prevent the delayed abort occurence all together by hiding the old handle h_1 returned from an oracle query (e.g. $\mathsf{DHEXP}(g^y, (g^x)^r)$ or $\mathsf{SET}_{dh}(h_2, 0, k_1)$). These proxied oracles in the package `Map` return the new expected handle (external view) to the adversay instead of the old handle stored in the key table when a mapping is triggered in the Log package. Moreover, the package `Map` retains the new to old handle mapping in a table so that it maps a new handle used by the adversary to the old handle (internal view) stored in its table to avoid the delayed aborts. For example, we reuse the variables from our example for infinity mapping behaviour of the Log package. Concretely, the proxied oracle DHEXP exposed by the package `Map` returns the handle h_2 (external view) to the adversary instead of h_1 (internal view) and maintains the mapping from h_2 to h_1 in the package table. The mapping information helps to replace the handle h_2 with h_1 and retrieve key k_1 from the key package when the adversary queries foro example $\mathsf{XTR}(h_{esalt}, h_2)$ in the future using the handle h_2. In other words, the adversary queries and receives external handles while external handles are mapped to internal handles by the `Map` package. As a result, internal handles are hidden from the adversary. Any adversary $\mathcal{A}$ against the game $\texttt{Gks}^{\texttt{0Map}}$ uses external handles but does not experience any aborts when calling the key derivation functions with colliding

DH handles. Observe that Figure 26 depicts the core key schedule security real game $\texttt{Gcore}^0$. We consider the figure as a definition and do not give an explict definition with sequential and parallel compositions separately. Any adversary against the game $\texttt{Gcore}^0$ uses internal handles and does experience oracle aborts when calling the key derivation functions with colliding handles. In Lemma C.2 of [BDLE+21], relying on Theorem 2.3, BDEFKK prove the code equivalence $\texttt{Gks}^0 \overset{code}{\equiv} \texttt{Gks}^{0\texttt{Map}}$ using a pen-and-paper invariant argument similar to proof of Claim 2.7. We take steps towards automatization of this proof with SSBee in Section 4. We present the state relations BDEFKK use as well as a few other state relations necessary to verify same-output proof obligations. We were also able to verify invariance of some these state relations with SSBee.

Before presenting the pseudocode of the `Map` package, we elaborate on the mapping parameter $map = 1$ for the package $\texttt{Log}^{A1}_{psk}$. The back propagation of D pattern also reaches the base key PSK. In other words, if the adversary is allowed to set the same PSK under two distinct dishonest PSK handles, there will be two distinct and dishonest handles for each of the output keys derived from the PSK. Notice that we prevented the adversary from setting an application PSK under two dishonest application PSK handles with the pattern A of the package $\texttt{Log}^{A}_{psk}$ in game $\texttt{Gks}^0$. However, as motivated in Section 3.1.1, the adversary may set an application PSK of a session with the resumption PSK of another session. This is possible in the security model by setting a dishonesy application PSK with a dishonest resumption PSK for some level computed by the adversary and the game. Therefore, it is preferred to abort on all dishonest PSK collisions (between dishoenst appliction and resumption PSKs) with the package $\texttt{Log}^{D}_{psk}$. However, in an approach similar to the choice of infinity mapping, BDEFKK choose to delay aborting the game and map the handle of the resumption PSK to the handle of the application PSK (or the other way around depending on which were set first in the game) with the *one-time* mapping $map = 1$. The difference between one-time and infinity mapping is that one-time mapping only maps a dishonest handle h_2 to dishonest handle h_1 only once for each key value and $\{\mathsf{level}(h_1), \mathsf{level}(h_2)\} = \{0, \ell\}$ where $\ell \neq 0$ (i.e. exactly one of them is a resumption PSK handle and the other one is an application PSK handle). In other words, the first time $\mathsf{UNQ}_{psk}(h_1, 0, k)$ is queried with key k, a Log entry $\mathrm{Log}[h_1] = (h_1, 0, k)$ is created. The second time $\mathsf{UNQ}_{psk}(h_2, 0, k)$ is queried with the same key k, the handle h_2 is mapped to h_1 a Log entry as $\mathrm{Log}[h_2] = (h_1, 0, k)$ and $J_{psk}[k] = 1$ is set. (Recall the key flag table J_n of the Log package.) The third time $\mathsf{UNQ}_{psk}(h_3, 0, k)$ is queried with the same key k, $J_{psk}[k] = 1$ and no mapping occurs but rather the pattern P condition is checked. As a result, when $P = D$, the game aborts on the third query but when $P = A$, the game aborts only if h_3 is an application PSK handle colliding with another application PSK handle. Notice, though, no mapping occurs if two dishonest resumption PSK handles share the same key. This is due to the fact that resumption PSKs are output of xpd operations and after having reduced to collision resistance of xpd, a collision of resumption PSKs transitively imply a collision of an application PSK with another application PSK or a resumption PSK. Moreover, one-time mapping of collisions between a resumption PSK and an application PSK

is enough. Let h_1 be a dishonest handle of a resumption PSK and h_2 be a dishonest handle of an application PSK such that they are both mapped to the same key k_{psk}. Without loss of generality, the Log table entries is as follows: $\mathrm{Log}[h_1] = (h_1, 0, k_{psk})$ and $\mathrm{Log}[h_2] = (h_1, 0, k_{psk})$. (i.e. h_2 is mapped to h_1) If $\mathsf{UNQ}_{psk}(h_3, 0, k)$ is queried to the package $\mathtt{Log}^{A1}_{psk}$ (or $\mathtt{Log}^{D1}_{psk}$) where $h_3 \neq h_2$ is a handle of the resumption PSK k_{psk} (i.e. a second collision), then h_3 and h_2 are two colliding resumption PSK violating the collision resistant xpd. Simialarly, if $\mathsf{UNQ}_{psk}(h_3, 0, k)$ is queried where $h_3 \neq h_1$ is a handle of the application PSK k_{psk}, then h_3 and h_1 are two colliding application PSK caught by the semantics of the A pattern. The last observation is crucial for the proof of Lemma C.2 as the adversary can not distinguish the package $\mathtt{Log}^{A}_{psk}$ being replaced with $\mathtt{Log}^{A1}_{psk}$. (We will show that the mapping does not prevent an abort of the A pattern.) We will get back to this point again in Section 4 when verifying the same-output and equal-aborts proof obligations for the exposed oracle $\mathsf{SET}_{psk,0}$. Analogous to infinity mapping, delaying game abort with one-time mapping is used by the proxy mapping package `Map` to hide the abort from the adversary.

We now describe how delaying the game abort can help hiding it from the adversary using the proxy package `Map`. Figure 27 shows the pseudocode of the package `Map`. The `Map` package exposes all the oracles exposed by the game $\mathtt{Gks}^0$. Observe that an oracle query $\mathsf{DHEXP}(X, Y)$ returns the external expected handle $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$ while stores the possibly mapped handle returned by the original DHEXP of the package `DH` in the table $M_{dh,\perp}$.

The oracle $\mathsf{XTR}(h_1, h_2)$ first retrieves the internal handles $M_{i_1}[h_1]$ and $M_{i_2}[h_2]$ of the external handles h_1 and h_2. In the next step, it computes the level ℓ' to call the oracle $\mathsf{XTR}_{n,\ell'}$ of the correct package $\mathtt{Xtr}_{n,\ell'}$. Exactly one of the given parent handles is leveled and maps to an internal handle with level level ℓ'. The level ℓ' is different from ℓ only if the input handles (external handles) are derived from a mapped PSK handle, not necessarily in the same level ℓ. Notice that $\mathsf{XTR}_{n,\ell}(h_1, h_2)$ calls $h' = \mathsf{XTR}_{n,\ell'}(M_{i_1}[h_1], M_{i_2}[h_2])$ with the internal handles to avoid abort. The mapping table helps to prevent abort with this delayed mechanisms., Essentially, internal handles are hidden from the adversary. Finally, the external handle $h = \mathsf{xtr}\langle n, h_1, h_2\rangle$ expected by the adversary is returned to the adversary and h is mapped to h' with $M_{n,\ell}[h] = h'$. Observe that if $M_{n,\ell}[h] \neq \perp$, then $n = \mathsf{name}(h)$ and $\mathsf{level}(h) = \ell$. As a result, the assertions in the beginning of the code ensure that h_1 and h_2 are the legitimate parent handles of newly constructed handle h. (i.e. they have the proper names and levels.) Looking forward to Section 4, this property can be proved as a *one-sided* state relation (state relation about the state of only one game, i.e. $\mathtt{Gks}^{0\mathrm{Map}}$) via induction because assuming the property holds for the given parent handles h_1 and h_2 (i.e. $\mathsf{level}(h_1) = \ell_1$ and $\mathsf{level}(h_2) = \ell_2$), we can show using the assignment $M_{n,\ell}[h] \leftarrow h'$, structure $\mathsf{xtr}\langle n, h_1, h_2\rangle$ of h, and recursive definition of level function that the property holds for h. We will give more examples of one-sided state relations and one-sided invariants in Section 4. We will also prove the invariant bubbling theorem for the one-sided invariants.

The oracle $\mathsf{XPD}(h_1, r, args)$ has a similar structure to $\mathsf{XTR}(h_1, h_2)$ by first retrieving the internal handle $M_{i_1}[h_1]$ of the given external parent handle h_1. Key

Map

State

$M_{n,\ell}$: mapping table

$\mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$

$M_{\mathrm{psk},0}[h] \leftarrow \mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$
return h

DHGEN()

return DHGEN()

$\mathsf{DHEXP}(X, Y)$

$h \leftarrow \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$
$h' \leftarrow \mathsf{DHEXP}(X, Y)$
$M_{dh,\perp}[h] \leftarrow h'$
return h

$\mathsf{XTR}_{n\in\{es,hs,as\},\ell}(h_1, h_2)$

$i_1, i_2 \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
assert $M_{i_2}[h_2] \neq \perp$
if $\mathsf{level}(M_{i_1}[h_1]) \neq \perp$:
  $\ell' \leftarrow \mathsf{level}(M_{i_1}[h_1])$
else :
  $\ell' \leftarrow \mathsf{level}(M_{i_2}[h_2])$
$h \leftarrow \mathsf{xtr}\langle n, h_1, h_2\rangle$
$h' \leftarrow \mathsf{XTR}_{n,\ell'}(M_{i_1}[h_1], M_{i_2}[h_2])$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{XPD}_{n\in XPN,\ell}(h_1, r, args)$

$i_1, \_ \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
$label \leftarrow \mathsf{Labels}(n, r)$
$\ell_1 \leftarrow \mathsf{level}(M_{i_1}[h_1])$
$h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
$h' \leftarrow \mathsf{XPD}_{n,\ell_1}(M_{i_1}[h_1], r, args)$
if $n = psk$ **then** $\ell \leftarrow \ell + 1$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{GET}_{n\in O^*,\ell}(h)$

assert $M_{n,\ell}[h] \neq \perp$
return $\mathsf{GET}_{n,\mathsf{level}(M_{n,\ell}[h])}(M_{n,\ell}[h])$

Helpers

$\mathsf{PrntIdx}(es, \ell)$

return $(0salt), (psk, \ell)$

$\mathsf{PrntIdx}(hs, \ell)$

return $(esalt, \ell), (dh)$

$\mathsf{PrntIdx}(as, \ell)$

return $(hsalt, \ell), (0ikm)$

$\mathsf{PrntIdx}(n, \ell)_{n\in N_{\mathsf{xpd}}}$

$(n_1, \_) \leftarrow \mathsf{PrntN}(n)$
return $(n_1, \ell), ()$

Figure 27: Package Map

derivation is invoked with the internal handle $M_{i_1}[h_1]$. Finally, a new external handle is constructed and is mapped to the internal handle computed by the key derivation oracle.

Missing from the work of [BDLE+21] is the initialization of the mapping table with the identity mapping of handles $\mathsf{noDH}\langle alg\rangle$, $\mathsf{noPSK}\langle alg\rangle$, $\mathsf{0ikm}\langle alg\rangle$, and 0salt. The reason for the mapping is the assertions in the beginning of the code of oracles XTR and XPD. These oracles expect given handles exist in the mapping table. These handles should all map to themselves as they all also map to themselves in the Log table. (e.g. $\mathrm{Log}_{dh}[\mathsf{noDH}\langle alg\rangle] = (\mathsf{noDH}\langle alg\rangle, 0, 0^{\mathsf{len}(alg)})$)

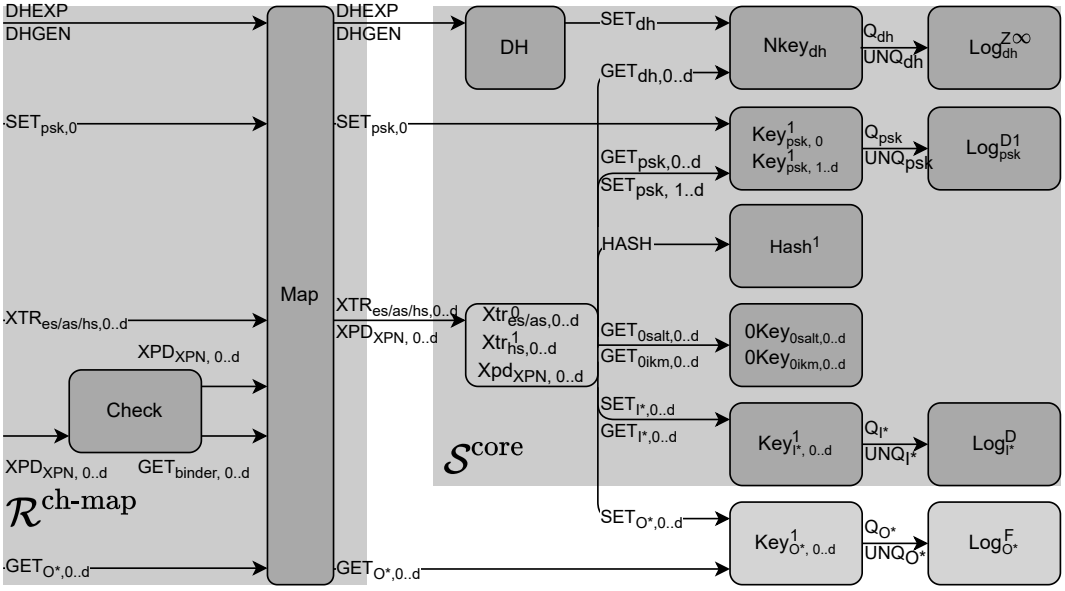


Figure 28: Game $\texttt{Gks}^{1\text{Map}}$ (copied with permission from [BDLE+21])

3.3.4 Applying the core key schedule security

Having prepared the scene, BDEFKK have ensured collision-free dishonest keys for the base keys (DH secrets and PSKs) with the Log packages $\texttt{Log}^{Z\infty}_{dh}$ and $\texttt{Log}^{A1}_{psk}$. Recall that $\texttt{Gks}^{0\text{Map}}$ is code equivalent to the composition of reduction package $\mathcal{R}^{\text{ch-map}}$ with the core key schedule real security game $\texttt{Gcore}^0$. (i.e. $\texttt{Gks}^{0\text{Map}} \overset{code}{\equiv} \mathcal{R}^{\text{ch-map}} \rightarrow \texttt{Gcore}^0$) BDEFKK define another hybrid game $\texttt{Gks}^{1\text{Map}}$ (visualized in figure 28). In a game hop from, they replace the game $\texttt{Gks}^{0\text{Map}}$ with $\texttt{Gks}^{1\text{Map}}$ by reducing (with reduction $\mathcal{R}^{\text{ch-map}}$) to the core key schedule security games $\texttt{Gcore}^b$.

3.3.5 Removing the mapping

As mentioned in Section 3.1.1, TLS 1.3 ensures uniqueness of the output keys by including the DH shares of the client and the server (in order) into the transcript and mixing the hash digest of the transcript when computing output keys (the last xpd derivation). Moreover, TLS 1.3 includes PSK binder value (modeled as the binder key k_{binder} in the security model) in the transcript (end of Client Hello message) for `psk_ke` or `psk_dh_ke` key exchange modes. The binder value itself is computed from k_{bind} using xpd, which is in turn computed indirectly from the PSK but with PSK type indicator labels `ext binder` or `res binder`. In any case, transcript is included in the very last xpd operation deriving each of the eight output keys. Despite being a complicated and twisted design, it allows to prove output key uniqueness. In Lemma C.5 of [BDLE+21], BDEFKK slightly modify the mapping package `Map` in order to remove the output key retrieval oracle $\mathsf{GET}_{O^*,[d]}$ from the mapping package `Map`. As a result, all calls to the oracles $\mathsf{GET}_{O^*,[d]}(h)$ are forwarded directly to the output key packages $\texttt{Key}^1_{O^*,[d]}$. Notice that the adversary queries $\texttt{Key}^1_{O^*,[d]}$ with external handles

of the output keys while random and unique output keys are stored under internal handles. BDEFKK modify the $\mathsf{XPD}_{O^*,[d]}$ oracles of the mapping package Map so that that for any external handle h_n of an output key name $n \in O^*$, $M_{n,\ell}[h_n] = h_n$ and the output key is stored under h_n. As a prerequisitive of this transformation, it is required that no two external handles map to the same internal handle. (i.e. an output key can not be stored under two distinct handles.) In Claim C.5.1 of [BDLE+21], BDEFKK prove the injectivity of the mapping table $M_{n,\ell}[h_n]$ for all external handles h_n of the output keys. They benefit from the transcript checks by the Check package in the injectivity proof. Recall the Check package ensured when deriving early output keys $\{eem, cet, binder\}$, the provided transcript contains the binder value corresponding to the provided handle. This ensures that level of the handle, which determines the type of PSK, has a correspondence with the transcript via the binder value. Moreover, the Check package made sure the DH shares in the handle structure are included in the transcript, which is hashed and used as an input for xpd. Both these checks prove the injectivity of handles for the output keys.

In order to set output keys under the external handles, they modify the code of $\mathsf{XPD}_{n,\ell}$ oracles computing the output keys, i.e. for all $n \in O^*$ and $\ell \in [d]$. However, the package Map does not set keys in the key packages but rather proxy all $\mathsf{XPD}_{n,\ell}$ oracle calls for all expand keys $n \in N_{\mathsf{xpd}}$ to the $\mathtt{Xpd}_{n,\ell}$ packages. BDEFKK modify the code of package Map and cosntruct a new package Map-Xpd by inlining code of oracles $\mathsf{XPD}_{O^*,[d]}$ of the packages $\mathtt{Xpd}_{O^*,[d]}$ so that Map-Xpd directly sets the output keys in the key packages $\mathtt{Key}^1_{O^*,[d]}$. In the next step, they modify the code of package Map-Xpd and cosntruct a new package Map-Xpd-Remap where they set output keys under the external handles (instead of internal handles) and remap the external handles of the output keys to achieve an identity mapping for them. We have presented the pseudocode of packages Map-Xpd and Map-Xpd-Remap in Figures 30 and 31.

Notice that in the oracle $\mathsf{XPD}_{n\in O^*,\ell}$ of package Map-Xpd-Remap, output keys are set under the external handle h derived from the input handle h_1 given by the adversary. Moreover, h is mapped to itself, i.e. $M_{n,\ell}[h] = h$.

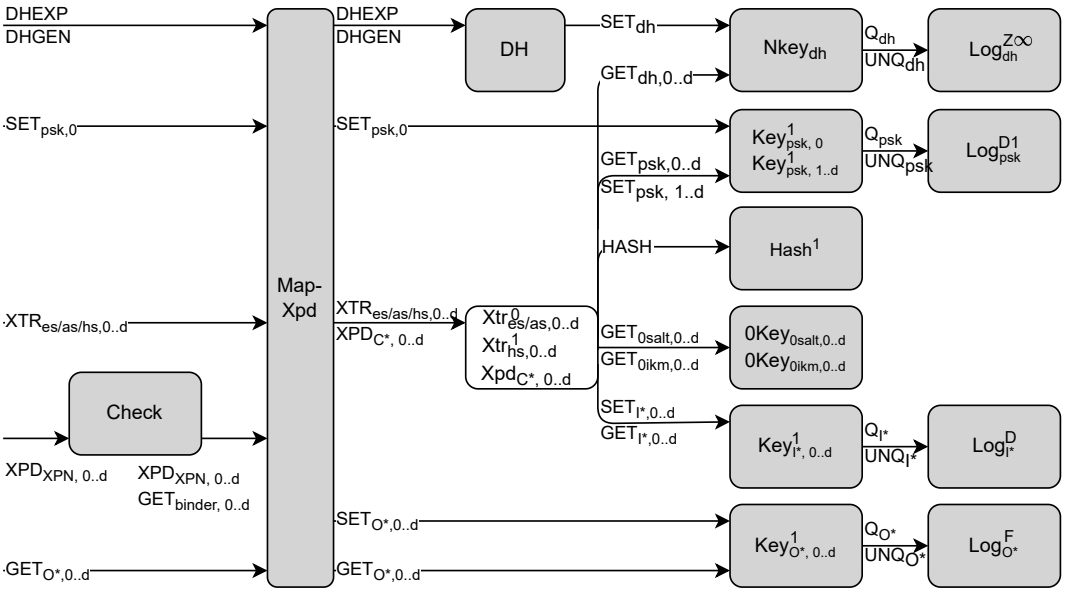
Formally, they define the hybrid games $\mathtt{Gks}^{\mathrm{MapXpd}}$ and $\mathtt{Gks}^1$ (visualized in Figures 29a and 29b with the shaded simulator package $\mathcal{S}$) using the modified mapping packages Map-Xpd and Map-Xpd-Remap, respectively. We consider the figures as the definition of these two hybrid games. The only difference of these two hybrid games with $\mathtt{Gks}^{1\mathsf{Map}}$ is the modified mapping packages.

BDEFKK prove the code equivalence $\mathtt{Gks}^{1\mathsf{Map}} \overset{code}{\equiv} \mathtt{Gks}^{\mathrm{MapXpd}}$ by simply inlining the code of oracles $\mathsf{XPD}_{n,\ell}$ of packages $\mathtt{Xpd}_{n,\ell}$ for all $n \in O^*$ and $\ell \in [d]$ in the package Map. Then, they show the code equivalence $\mathtt{Gks}^{\mathrm{MapXpd}} \overset{code}{\equiv} \mathtt{Gks}^1$ in Lemma C.5 of [BDLE+21]. BDEFKK complete the proof of Theorem C.1 by letting the simulator $\mathcal{S}$ (as shaded in Figure 29b) be the composition package of all the packages in the game $\mathtt{Gks}^1$ except for packages $\mathtt{Key}^1_{O^*,[d]}$ and $\mathtt{Log}_{O^*}$.

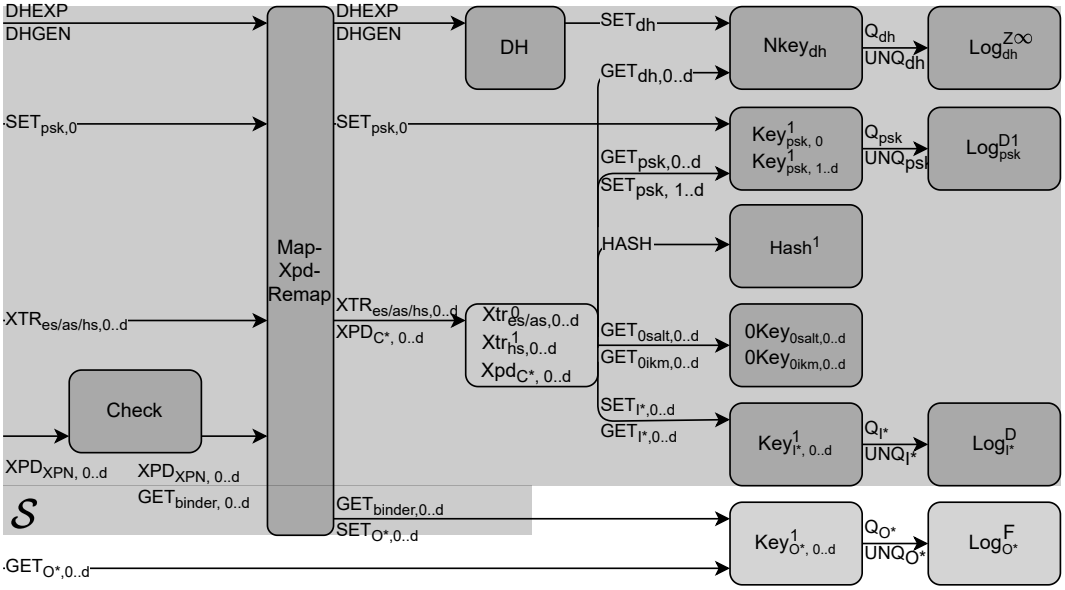
In a nutshell, proof of Theorem C.1 consists of five game hops:

$$\mathtt{Gks}^0 \overset{\text{Lemma C.2}}{\equiv} \mathtt{Gks}^{0\mathsf{Map}} \overset{\text{Reduction to } \mathtt{Gcore}^b}{\approx} \mathtt{Gks}^{1\mathsf{Map}} \overset{\text{code inlining}}{\equiv} \mathtt{Gks}^{\mathrm{MapXpd}} \overset{\text{Lemma C.5}}{\equiv} \mathtt{Gks}^1.$$

where $\mathtt{Gks}^{b\mathsf{Map}} \overset{code}{\equiv} \mathcal{R}^{\text{ch-map}} \rightarrow \mathtt{Gcore}^b$ and $\mathtt{Gcore}^0 \overset{\text{Hybrid argument}}{\approx} \mathtt{Gcore}^1$.



(a) Game $\texttt{Gks}^{\text{MapXpd}}$: $C^* := N_{\text{xpd}} \setminus O^*$ is the set of all expand keys except the output keys



(b) Game $\texttt{Gks}^1$

Figure 29: Games $\texttt{Gks}^{\text{MapXpd}}$ and $\texttt{Gks}^1$ (copied with permission from [BDLE+21])

Map-Xpd

State

$M_{n,\ell}$: mapping table

$\mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$

$M_{\mathrm{psk},0}[h] \leftarrow \mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$
return h

DHGEN()

return DHGEN()

$\mathsf{DHEXP}(X, Y)$

$h \leftarrow \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$
$h' \leftarrow \mathsf{DHEXP}(X, Y)$
$M_{dh,\perp}[h] \leftarrow h'$
return h

$\mathsf{XTR}_{n\in\{es,hs,as\},\ell}(h_1, h_2)$

$i_1, i_2 \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
assert $M_{i_2}[h_2] \neq \perp$
if $\mathsf{level}(M_{i_1}[h_1]) \neq \perp$:
  $\ell' \leftarrow \mathsf{level}(M_{i_1}[h_1])$
else :
  $\ell' \leftarrow \mathsf{level}(M_{i_2}[h_2])$
$h \leftarrow \mathsf{xtr}\langle n, h_1, h_2\rangle$
$h' \leftarrow \mathsf{XTR}_{n,\ell'}(M_{i_1}[h_1], M_{i_2}[h_2])$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{XPD}_{n\in N_{\mathsf{xpd}}\setminus O^*,\ell}(h_1, r, args)$

$i_1, \_ \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
$label \leftarrow \mathsf{Labels}(n, r)$
$\ell_1 \leftarrow \mathsf{level}(M_{i_1}[h_1])$
$h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
$h' \leftarrow \mathsf{XPD}_{n,\ell_1}(M_{i_1}[h_1], r, args)$
if $n = psk$ **then** $\ell \leftarrow \ell + 1$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{XPD}_{n\in O^*,\ell}(h_1, r, args)$

$i_1, \_ \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
$label \leftarrow \mathsf{Labels}(n, r)$
$\ell_1 \leftarrow \mathsf{level}(M_{i_1}[h_1])$
$h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
$n_1, \_ \leftarrow \mathsf{PrntN}(n)$
$h' \leftarrow \mathsf{xpd}\langle n, label, M_{i_1}[h_1], args\rangle$
$(k_1, hon) \leftarrow \mathsf{GET}_{n_1,\ell_1}(M_{i_1}[h_1])$
$d \leftarrow \mathsf{HASH}(args)$
$k \leftarrow \mathsf{xpd}(k_1, (label, d))$
$h' \leftarrow \mathsf{SET}_{n,\ell}(h', hon, k)$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{GET}_{n\in O^*,\ell}(h)$

assert $M_{n,\ell}[h] \neq \perp$
return $\mathsf{GET}_{n,\mathsf{level}(M_{n,\ell}[h])}(M_{n,\ell}[h])$

Figure 30: Package Map-Xpd

`Map-Xpd-Remap`

State

$M_{n,\ell}$: mapping table

$\mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$

$M_{\mathrm{psk},0}[h] \leftarrow \mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$
return h

DHGEN()

return DHGEN()

$\mathsf{DHEXP}(X, Y)$

$h \leftarrow \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$
$h' \leftarrow \mathsf{DHEXP}(X, Y)$
$M_{dh,\perp}[h] \leftarrow h'$
return h

$\mathsf{XTR}_{n\in\{es,hs,as\},\ell}(h_1, h_2)$

$i_1, i_2 \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
assert $M_{i_2}[h_2] \neq \perp$
if $\mathsf{level}(M_{i_1}[h_1]) \neq \perp$:
  $\ell' \leftarrow \mathsf{level}(M_{i_1}[h_1])$
else :
  $\ell' \leftarrow \mathsf{level}(M_{i_2}[h_2])$
$h \leftarrow \mathsf{xtr}\langle n, h_1, h_2\rangle$
$h' \leftarrow \mathsf{XTR}_{n,\ell'}(M_{i_1}[h_1], M_{i_2}[h_2])$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{XPD}_{n\in N_{\mathsf{xpd}}\setminus O^*,\ell}(h_1, r, args)$

$i_1, \_ \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
$label \leftarrow \mathsf{Labels}(n, r)$
$\ell_1 \leftarrow \mathsf{level}(M_{i_1}[h_1])$
$h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
$h' \leftarrow \mathsf{XPD}_{n,\ell_1}(M_{i_1}[h_1], r, args)$
if $n = psk$ **then** $\ell \leftarrow \ell + 1$
$M_{n,\ell}[h] \leftarrow h'$
return h

$\mathsf{XPD}_{n\in O^*,\ell}(h_1, r, args)$

$i_1, \_ \leftarrow \mathsf{PrntIdx}(n, \ell)$
assert $M_{i_1}[h_1] \neq \perp$
$label \leftarrow \mathsf{Labels}(n, r)$
$\ell_1 \leftarrow \mathsf{level}(M_{i_1}[h_1])$
$h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
$n_1, \_ \leftarrow \mathsf{PrntN}(n)$

$(k_1, hon) \leftarrow \mathsf{GET}_{n_1,\ell_1}(M_{i_1}[h_1])$
$d \leftarrow \mathsf{HASH}(args)$
$k \leftarrow \mathsf{xpd}(k_1, (label, d))$
$h \leftarrow \mathsf{SET}_{n,\ell}(h, hon, k)$
$M_{n,\ell}[h] \leftarrow h$
return h

$\mathsf{GET}_{n\in O^*,\ell}(h)$

assert $M_{n,\ell}[h] \neq \perp$
return $\mathsf{GET}_{n,\mathsf{level}(M_{n,\ell}[h])}(M_{n,\ell}[h])$

Figure 31: Package `Map-Xpd-Remap`: Red and empty lines show changes from `Map-Xpd`

Proof of Lemma C.5, again, relies on Theorem 2.3 and is a pen-and-paper invariant argument. Interestingly, BDEFKK express the injectivity property as a state relation among a few other state relations. They prove Claim C.5.1 by showing the invariance of their state relations. In this thesis, we only focus on formal verification of proof obligations for Lemma C.2 and leave automatization of Lemma C.5 to a future work.

Remark. BDEFKK suggest in their work that TLS 1.3 standard be updated to extract from DH secret together with DH shares immediately before mixing other keying material for easier security reduction and more well-understood assumption (Oracle Diffie-Hellman assumption by [ABR01] instead of Salted Oracle Diffie-Hellman assumption analyzed in Section 5). Similarly, they recommend that the TLS 1.3 standard extracts from PSK values and their types (application or resumption) in the beginning to make all keys collision-free and unique instead of only using `ext/res binder` label for the binder value computation and indirectly including binder values in the transcript used by output keys. These changes significantly reduce their reduction complexity, eliminating mapping lemmata (Lemma C.2 and Lemma C.5) and Appendix C altogether.

4 Towards Formal Verification of Key Schedule Security in SSBee

In this section, we discuss our verification approach to the TLS 1.3 key schedule security reduction introduced in Section 3. We demonstrate some corrections to the security model and additional lemmata necessary for verification. Furthermore, we prove the invariant bubbling theorem and illustrate how it simplifies the verification. Finally, we conclude with a a glossary of verification techniques in SSBee discovered during this project.

As discussed in Section 3.3.3, Lemmata C.2 and C.5 of [BDLE+21] are the major and most challenging code equivalence steps in the TLS 1.3 key schedule security reduction. Relying on Theorem 2.3, BDEFKK have presented a set of state relations and proved the same-output property and invariance of state relations on paper. Since the key schedule security games consist of several packages with many lines of code, their proof has spanned more than 10 pages. As mentioned in Section 2.5, SSBee tries to automate this process and reduce the amount of proof work as soon as a reasonable set of state relations are determined as stated for SSBee. In this thesis, we focus on verifying the code equivalence presented in Lemma C.2 with SSBee and defer verification of Lemma C.5 as well as the injectivity property to a future work. Our work demonstrates that automatization of code equivalence proofs in large scale cryptographic analysis for real world protocols (like TLS) with complex security games is possible. However, we point out difficulties still present and future directions for SSBee.

Recall that Lemma C.2 shows the code equivalence of the games $\texttt{Gks}^{0}$ and $\texttt{Gks}^{\texttt{0Map}}$. Moreover, we saw in Section 2.5 that the first step for verification of a code equivalence between two games in SSBee is to define the security games, their packages, and at least the state relations between the games. Furthermore, some code equivalences also need additional lemmata to be proved. We use the state relations presented by BDEFKK and introduce new relations and auxiliary lemmata when necessary. We begin with translating the pseudocode of packages composed in the games $\texttt{Gks}^{0}$ and $\texttt{Gks}^{\texttt{0Map}}$ to the SSBee language. Due to the complexity and size of the games, we generate the composition of packages with a script that directly outputs SSBee composition files based on the key schedule security games topology. The script, written in Python, also outputs the skeleton of the proof file that helps with iterative and agile proof development by extracting required information (such as abstract function package parameters) from the code of packages and automating generation of otherwise boilerplate code. Relying on Theorem 2.3, we then proceed with proving the verification obligations (same-output and equal-aborts properties, and invariance of the state relations) for each oracle exposed by the games. Hereafter, for ease of referring to the games $\texttt{Gks}^{0}$ and $\texttt{Gks}^{\texttt{0Map}}$, we call them left and right games, respectively.

4.1 Translation of games and packages pseudocode to SSBee language

All packages in the left and right games except DH, Check, and Map are parameterized with a name or level or both. There are at least two approaches to translate the packages into SSBee language. One approach generates SSBee packages parameterized by names and levels. This requires instantiating packages with concrete names and levels for a concrete number of levels d when defining the security games in composition files. (See Section 2.5 for a discussion on the difference of SSP packages and SSBee packages as well as the concept of package instances in SSBee.) That is, one has to choose some d (say $d = 3$) for the maximum number of session resumption levels, and instantiate all packages (for example $\mathsf{Key}^b_{n,\ell}$) for all names $n \in N \setminus \{dh, 0salt, 0ikm\}$ and levels $\ell \in \{0, 1, 2, 3\}$. Therefore, the size of composition files (games) linearly grows with the number of resumption levels. Precisely, since $|N| = 18$, there are 16 Log packages Log_n for $n \in N \setminus \{0salt, 0ikm\}$, 15 Key packages $\mathsf{Key}^b_{n,\ell}$ in each resumption level ℓ for $n \in N \setminus \{dh, 0salt, 0ikm\}$ and 3 non-leveled global NKey packages NKey_n shared among all levels for $n \in \{dh, 0salt, 0ikm\}$, 14 Xpd packages $\mathsf{Xpd}_{n,\ell}$ in level $\ell = 0$ and 15 packages in level $\ell > 0$ for $n \in N_{\mathsf{xpd}}$ (psk is set directly as application PSK in the first level but derived as a resumption PSK in higher levels), 3 Xtr packages $\mathsf{Xpd}_{n,\ell}$ in each level for $n \in \{es, as, hs\}$, 3 packages DH, Check, and Hash shared among all levels. In total, there are 22 global packages and 33 leveled packages; hence, $22 + 33d$ packages for d resumption levels. Presumably, compositions are written by an SSBee user and this growing number of package instances quickly go out of control even for small d. We automate generation of the composition files with a Python script. The script receives the number of resumption levels d and generates the composition files for the security games Gks^0, $\mathsf{Gks}^{0\mathrm{Map}}$, $\mathsf{Gks}^{\mathrm{MapXpd}}$, $\mathsf{Gks}^1(\mathcal{S})$ by instantiating all the necessary packages and constructing the call graph. Briefly, the script uses a stack-based algorithm and starting from a stack with a single package *adversary*, it pops a package from the stack, instantiates the packages with appropriate concrete parameters (including function parameters), and pushes the dependencies of the package to the stack. At the same time, it completes the call graph one package at a time by adding edges and nodes for the dependencies of the popped package. Another useful feature of the script is parsing package files [18] and extracting function parameters signatures and automatically propagating them to the composition and proof files. The script helped us to focus on translating package pseudocode in [BDLE+21] to SSBee language and delegating generation of the boilerplate code of compositions and the proof file function parameter definitions to the script. However, on each invocation, SSBee took a long time (more than an hour) to compile our packages, compositions and proof files to SMT-LIB code, only to then point out some parse errors. Since modifying the package codes as well as adding invariants and lemmata are the main parts of the iterative formal verification process, we decided to change our translation approach to speed up the process. The root cause of the issue is the current

[18] We partially rewrite the SSBee language grammar—originally written in Parsing Expression Grammar (PEG) syntax for the Pest parser library in Rust—in the Extended Backus-Naur form (EBNF) syntax for the Lark parser library in Python.

implementation and architecture of SSBee. One observation is that SSBee currently generates an SMT-LIB function for each oracle of each package [19] instance. This is due to the fact that different package parameters can generate different SMT-LIB codes for the oracles of the packages. Optimally, one would want to generate a template SMT-LIB code for the oracles with placeholders for the parameters and then plugin the correct concrete parameters for each package instance. However, this brings other complexities that we prefer to leave it for a future investigations and optimizations of SSBee for proofs with large compositions. Nevertheless, this project brought up such an issue for the development team of SSBee. The SSBee project files and the Python script of this approach are available online on our GitHub Repository [Raj25c] under the directory `tls13-key-schedule` in the example projects directory.

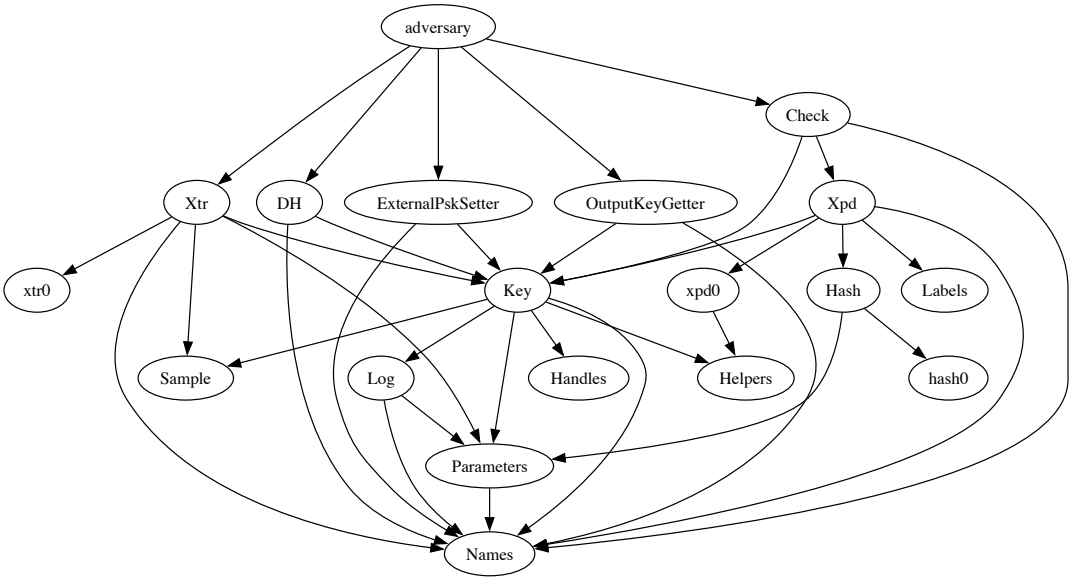
The other translation approach is to remove package parameters and add name and level as new arguments of the oracles. From the three leveled packages, the key derivation packages $\texttt{Xtr}^{b}_{n,\ell}$ and $\texttt{Xpd}_{n,\ell}$ are stateless and can be easily adapted by adding name and level as new arguments for their oracles, resulting in parameter-free packages `Xtr` and `Xpd`. We will see how the idealization bit b is also dropped. However, removing name and level parameters from the leveled Key package collapses all such packages and merge their key tables. Although it can be easily proved that the key table $K_{n,\ell}$ stores handles h such that $\mathsf{name}(h) = n$ and $\mathsf{level}(h) = \ell$, which implies that the tables $K_{n,\ell}$ have disjoint domains and supports the safe merge of the tables, the first line of the oracle $\mathsf{GET}_{n,\ell}(h)$ exposed by the key package checks that the given handle h already exists in the table $K_{n,\ell}$ by asserting $\textbf{assert}\ K_{n,\ell}[h] \neq \bot$. This assertion implicitly checks the name and level of the handle h, discouraging any simple merge of the tables such as $K[h]$. Nevertheless, we refrain from explicitly adding such assertions to the code of the oracle $\mathsf{GET}_{n,\ell}(h)$ and keep the same semantics by mapping tuples of (n, ℓ, h) to the keys (and their honesty bits) in the merged table K, i.e. $K_{n,\ell}[h]$ is syntactically replaced with $K[(n, \ell, h)]$. We even take one step further and use the same idea to collapse all non-leveled Log packages into one package and merge the Log tables and syntactically replace all appearances of $\mathrm{Log}_n[h]$ with $\mathrm{Log}[(h, h)]$. Notice that the same scenario is true for the Log tables Log_n and they store handles such that $\mathsf{name}(h) = n$ (i.e. their domains are disjoint) but the oracle $\mathsf{Q}_n(h)$ checks for the existence of the handle h in the Log table.

Apart from the names and levels, packages are instantiated with other parameters such as function parameters, idealization bits, patterns, and mapping parameters (for the Log packages). Notice that the idealization bit of the Key and Xtr packages are a function of name, level, and the game, i.e. $\texttt{Gks}^{0}$, $\texttt{Gks}^{0\mathrm{Map}}$, etc. Similarly, the pattern and mapping parameters of the Log package depends on the name and the game while the idealization bit of the Hash packages depends on only the game. These observations were inspired by our Python script implementation. Therefore, we introduce an auxiliary package `Parameters` that provide the collapsed packages with these parameters but parametrized with the game. We will see the SSBee code of translated packages and other auxiliary packages shortly. The SSBee project files of

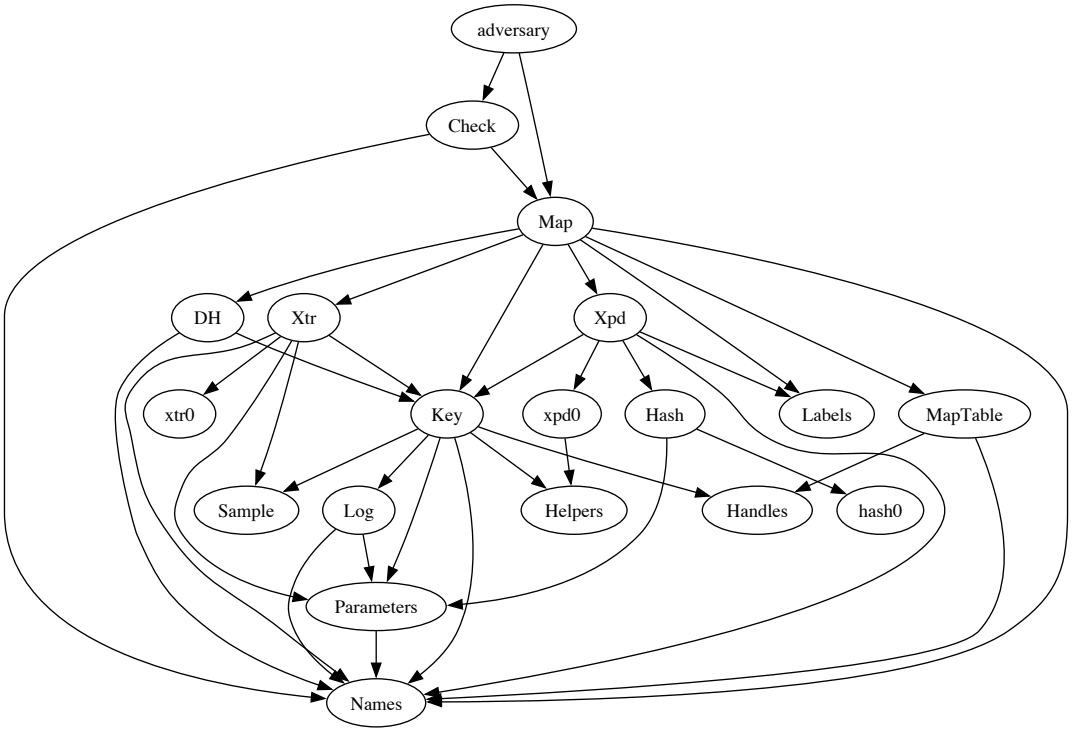
[19] Precisely, the oracles of packages that are called directly or indirectly (recursively when inlined) by the oracles exposed to the adversary.

this approach are available online on the same GitHub Repository [Raj25b] under the directory `tls13-key-schedule-parameterless` in the example projects directory.

Before then, we want to compare the translation approaches. The first advantage regards the resumption levels. The second approach allows us to verify the code equivalence of the games for any number of resumption levels while the first approach is essentially limited to the number of levels d for which we have generated package instances. Notice that SSBee does not support dynamic package instantiation in the composition files because it does not support any iteration control flow statement in the composition files. The second advantage considers the next step of the verification, i.e. code equivalence proof. As mentioned earlier, Theorem 2.3 requires invariant state relations. We saw in Section 2.5 that state relations should be expressed with SMT-LIB language in a separate file and then referenced in the proof file. These state relations express some relations between the states of the packages in the left and right games; hence, they need to destructure the game states and extract tables and variables from the relevant package states. SSBee internally generates a new SMT-LIB data type for each package instance used in a game and naturally creates a new field for the state of each package instance in the game state. As we will see the state relations in this section, many state relations common to all Key packages for all key names and in all resumption levels. Expressing such state relations needs to state the state relations for each name and level as a separate formula. Since the data types are syntactically different in the SMT-LIB language, one can not use a universal quantifier. As a result, the invariant file would need to be automatically generated which reduces its readability. Speaking of readability, the second approach makes the composition files concise and more readable although we have adapted our Python script to automatically generate them and facilitate the function parameters propagation. Formalization of the game $\mathtt{Gks}^0$ with the second approach consists of less than 300 lines of code for arbitrary number of resumption levels compared to the over 2000 lines of code for only 1 resumption level. On the other hand, the second approach still hides the packages compositions from the reader, similar to the concise call graph in Figure 21a compared to the preceding sequential-and-parallel-composition-based definition. We, therefore, encourage the reader to run the Python script themselves or investigate the repository of the first approach in which the TLS 1.3 security games are generated for 3 resumption levels for the extended call graph of the games. Apart from the hidden game composition, the second has an important disadvantage of blocking verification of the core key schedule theorem. Recall from Section 3.3.2 that the code key schedule theorem is proved via a hybrid argument and SSP-style reductions as graph cuts to modular assumptions. The graph cuts are possible due to the separation of key tables of the Key packages for different key names and levels. When all Key, Log, Xtr, and Xpd packages are collapsed to only one package, it is not possible to verify the graph cuts with SSBee. This is unfortunate as SSBee is designed to easily express and verify SSP-style reductions and graph cuts. Thus, we leave the verification of the core key schedule theorem in SSBee to a future work. We speculate that SSBee suboptimal SMT-LIB translation issue mentioned earlier should not interfere with reduction verification because no SMT-LIB translation is necessary and graph cuts are checked algorithmically. However, we can't confirm this statement or predict other



(a) Composition `Gks0` (formalized $\texttt{Gks}^{0}$)



(b) Composition `Gks0Map` (formalized $\texttt{Gks}^{0\text{Map}}$)

Figure 32: Call graphs of compositions `Gks0` and `Gks0Map`

possible complications as we did not look into this problem in this thesis.

4.1.1 Game compositions

Figures 32a and 32b shows the call graphs of the games $\texttt{Gks}^{0}$ and $\texttt{Gks}^{0\text{Map}}$ as compositions of our collapsed packages and auxiliary packages. We adapt our Python script from the first approach to generate composition file together with the call graph diagrams.

Notice the auxiliary packages `ExternalPskSetter`, `OutputKeyGetter`, `xtr0`, `xpd0`, `hash0`, `Names`, `Handles`, `Labels`, `Helpers`, and `Parameters`. Observe that packages `Xtr`, `Key`, `Log`, and `Hash` depend on the package `Parameters` for the idealization bits and other parameters.

```
composition Gks0 {
    ...
    instance pkg_Key = Key {
        ...
    }
    ...
    instance pkg_Parameters = Parameters {
        params {
            game: 0,
            ...
        }
    }
    ...
    compose {
        ...
        pkg_Log: {
            GET_LOG_PACKAGE_PARAMETERS: pkg_Parameters,
            ...
        },
        pkg_Key: {
            GET_KEY_PACKAGE_IDEALIZATION_PARAMETER: pkg_Parameters,
            Q: pkg_Log,
            UNQ: pkg_Log,
            ...
        },
        pkg_ExternalPskSetter: {
            GET_PSK_NAME: pkg_Names,
            SET: pkg_Key,
        },
        pkg_OutputKeyGetter: {
            GET: pkg_Key,
            IS_OUTPUT_KEY: pkg_Names,
        },
        ...
        adversary: {
            DHEXP: pkg_DH,
            DHGEN: pkg_DH,
            GET: pkg_OutputKeyGetter,
            SET: pkg_ExternalPskSetter,
            XPD: pkg_Check,
            XTR: pkg_Xtr,
        }
```

```
    }
}
```

Listing 3: Game $\texttt{Gks}^0$ in SSBee

```
composition Gks0Map {
    ...
    pkg_Map: {
        DHEXP: pkg_DH,
        DHGEN: pkg_DH,
        GET: pkg_Key,
        GETMAP: pkg_MapTable,
        GET_DH_NAME: pkg_Names,
        GET_PSK_NAME: pkg_Names,
        IS_OUTPUT_KEY: pkg_Names,
        IS_PSK: pkg_Names,
        IS_XPD_KEY: pkg_Names,
        IS_XTR_KEY: pkg_Names,
        LABEL: pkg_Labels,
        PARENTS: pkg_Names,
        SET: pkg_Key,
        SETMAP: pkg_MapTable,
        XPD: pkg_Xpd,
        XTR: pkg_Xtr,
    },
    ...
    instance pkg_Parameters = Parameters {
        params {
            game: 1,
            ...
        }
    }
    ...
    compose {
        ...
        adversary: {
            DHEXP: pkg_Map,
            DHGEN: pkg_Map,
            GET: pkg_Map,
            SET: pkg_Map,
            XPD: pkg_Check,
            XTR: pkg_Map,
        },
    }
}
```

Listing 4: Game $\texttt{Gks}^0$ in SSBee

Listing 3 shows part of the composition file for the formalization of the left game `Gks0` in SSBee. Listing 4 shows part of the composition file for the right game `Gks0Map`. Observe that in the left game, `Parameters` package is instantiated with the game parameter 0, encoding $\texttt{Gks}^0$, while in the right game it is instantiated with game parameter 1, encoding $\texttt{Gks}^{0\text{Map}}$. Listing 5 shows the code of package `Parameters`. Notice how we have delegated computation of the value of the pattern and mapping

parameters of the Log package to an abstract function. In Listing 6 we specify the value of the abstract function at desired points in the SMT-LIB code. This code is later on references by the proof file. Since SSBee becomes very slow when generating SMT-LIB code if the code contains nested if-conditions, we delegate switch-case style checks to abstract functions as part of out optimizations.

```
package Parameters {
    params {
        /*
            0: Gks0,
            1: Gks0Map,
            2: GksMapXpd,
            3: Gks1
        */
        game: Integer,
        log_package_parameters: fn Integer, Bool, Bool, Bool, Bool -> (Integer, Integer),
    }

    import oracles {
        IS_OUTPUT_KEY(n: Integer) -> Bool,
        IS_INTERNAL_KEY(n: Integer) -> Bool,
        IS_DH_KEY(n: Integer) -> Bool,
        IS_PSK(n: Integer) -> Bool,
        IS_HANDSHAKE_SECRET(n: Integer) -> Bool,
    }

    oracle GET_HASH_PACKAGE_IDEALIZATION_PARAMETER() -> Bool {
        if ((game == 0) or (game == 1)) { /* before idealization */
            return false;
        }
        if ((game == 2) or (game == 3)) { /* after idealization */
            return true;
        }
        abort;
        /* This should never happen and we put it here to make "function" total */
        return false;
    }

    oracle GET_KEY_PACKAGE_IDEALIZATION_PARAMETER(n: Integer, l: Integer) -> Bool {
        is_dh <- invoke IS_DH_KEY(n);
        is_psk <- invoke IS_PSK(n);
        if is_dh {
            return false;
        }
        if ((game == 0) or (game == 1)) { /* before idealization */
            if is_psk {
                if (l == 0) {
                    return true;
                }
                return false;
            }
            return false;
        }
        if ((game == 2) or (game == 3)) { /* after idealization */
```

```
            return true;
        }
        abort;
        /* This should never happen and we put it here to make "function" total */
        return false;
    }

    oracle GET_XTR_PACKAGE_IDEALIZATION_PARAMETER(n: Integer, l: Integer) -> Bool {
        if ((game == 0) or (game == 1)) { /* before idealization */
            return false;
        }
        if ((game == 2) or (game == 3)) { /* after idealization */
            is_hs <- invoke IS_HANDSHAKE_SECRET(n);
            if is_hs {
                return true;
            }
            return false;
        }
        abort;
        /* This should never happen and we put it here to make "function" total */
        return false;
    }

    oracle IS_INFINITY_MAPPING(mapping: Integer) -> Bool {
        if (mapping == 2) {
            return true;
        }
        return false;
    }

    oracle IS_1_MAPPING(mapping: Integer) -> Bool {
        if (mapping == 1) {
            return true;
        }
        return false;
    }

    oracle IS_A_PATTERN(pattern: Integer) -> Bool {
        if (pattern == 1) {
            return true;
        }
        return false;
    }

    oracle IS_D_PATTERN(pattern: Integer) -> Bool {
        if (pattern == 2) {
            return true;
        }
        return false;
    }

    oracle IS_F_PATTERN(pattern: Integer) -> Bool {
        if (pattern == 3) {
            return true;
```

```
        }
        return false;
    }

    oracle GET_LOG_PACKAGE_PARAMETERS(n: Integer) -> (Integer, Integer) {
        is_dh <- invoke IS_DH_KEY(n);
        is_psk <- invoke IS_PSK(n);
        is_internal <- invoke IS_INTERNAL_KEY(n);
        is_output <- invoke IS_OUTPUT_KEY(n);
        if (not is_dh and not is_psk and not is_internal and not is_output) {
            abort;
        }
        (pattern, mapping) <- parse log_package_parameters(game, is_dh, is_psk,
    is_internal, is_output);
        return (pattern, mapping);
    }
}
```

Listing 5: Package `Parameters`

```
; DH (Gks0) = Z

(assert
    (=
        (<<func-log_package_parameters>> 0 true false false false) ; Gks0 is 0, is_dh =
    true, is_psk = is_internal = is_output = false
        (mk-tuple2 0 0) ; pattern = 0 (Z) and mapping = 0
    )
)

; DH (Gks0Map) = Zinf

(assert
    (=
        (<<func-log_package_parameters>> 1 true false false false) ; Gks0Map is 1, is_dh
    = true, is_psk = is_internal = is_output = false
        (mk-tuple2 0 2) ; pattern = 0 (Z) and mapping = infinity
    )
)

; PSK (Gks0) = A

(assert
    (=
        (<<func-log_package_parameters>> 0 false true false false) ; Gks0 is 0, is_dh =
    false, is_psk = true, is_internal = is_output = false
        (mk-tuple2 1 0) ; pattern = 1 (A) and mapping = 0
    )
)

; PSK (Gks0Map) = A1

(assert
    (=
```

```
        (<<func-log_package_parameters>> 1 false true false false) ; Gks0Map is 1, is_dh = false, is_psk = true, is_internal = is_output = false
        (mk-tuple2 1 1) ; pattern = 1 (A) and mapping = 1
    )
)

; Internal keys (Gks0) = Z

(assert
    (=
        (<<func-log_package_parameters>> 0 false false true false) ; Gks0 is 0, is_dh = false, is_psk = false, is_internal = true, is_output = false
        (mk-tuple2 0 0) ; pattern = 0 (Z) and mapping = 0
    )
)

; Internal keys (Gks0Map) = Z

(assert
    (=
        (<<func-log_package_parameters>> 1 false false true false) ; Gks0Map is 1, is_dh = false, is_psk = false, is_internal = true, is_output = false
        (mk-tuple2 0 0) ; pattern = 0 (Z) and mapping = 0
    )
)

; Output keys (Gks0) = Z

(assert
    (=
        (<<func-log_package_parameters>> 0 false false false true) ; Gks0 is 0, is_dh = false, is_psk = false, is_internal = false, is_output = true
        (mk-tuple2 0 0) ; pattern = 0 (Z) and mapping = 0
    )
)

; Output keys (Gks0Map) = Z

(assert
    (=
        (<<func-log_package_parameters>> 1 false false false true) ; Gks0Map is 1, is_dh = false, is_psk = false, is_internal = false, is_output = true
        (mk-tuple2 0 0) ; pattern = 0 (Z) and mapping = 0
    )
)
```

Listing 6: Log package parameters abstarct function definition

Recall that in $\texttt{Gks}^0$ the adversary was given access directly to the output key packages $\texttt{Key}^0_{n\in O^*,\ell}$ and the application PSKs key package $\texttt{Key}^1_{psk,0}$. Since all key packages (including `NKey`, `0salt`, and `0ikm`) are collapsed into one `Key` package, we can not give direct access to the package setter and getter. Instead we proxy these calls to the package `Key` through `ExternalPskSetter` and `OutputKeyGetter` to ensure only

application PSKs are set and output keys are retrieved. Listing 7 shows the code of packages `ExternalPskSetter` and `OutputKeyGetter` in SSBee.

```
package OutputKeyGetter {
    import oracles {
        GET(n: Integer, l: Integer, h: Bits(*)) -> (Bits(*), Bool),
        IS_OUTPUT_KEY(n: Integer) -> Bool,
    }

    oracle GET(n: Integer, l: Integer, h: Bits(*)) -> (Bits(*), Bool) {
        is_output_key <- invoke IS_OUTPUT_KEY(n);
        if not is_output_key {
            abort;
        }
        t <- invoke GET(n, l, h);
        return t;
    }
}
package ExternalPskSetter {
    import oracles {
        SET(n: Integer, l: Integer, h: Bits(*), hon: Bool, k: Bits(*)) -> Bits(*),
        GET_PSK_NAME() -> Integer,
    }

    oracle SET(h: Bits(*), hon: Bool, k: Bits(*)) -> Bits(*) {
        psk <- invoke GET_PSK_NAME();
        t <- invoke SET(psk, 0, h, hon, k);
        return t;
    }
}
```

Listing 7: Code of packages `ExternalPskSetter` and `OutputKeyGetter` in SSBee

Notice that oracles `GET` and `SET` of the key package `pkg_Key` receives name and level in addition to their other arguments. We use integers to encode names, although SSBee supports strings but it does not have support for string literals yet which makes handle construction and comparison difficult. We use `Bits(*)` for handles. One can think of it as the bit encoding of the handle data structure. As we will see in the code of packages, we define abstract functions to construct DH, extract and expand handles. Moreover, abstract functions can be used to retrieve various fields from the handles such as DH shares, parent handles, names, transcript, and handle types. Abstract functions can also be used for handle level computation. The same idea can be used for any data structure that is not yet supported by SSBee. With respect to encodings, bitstrings of arbitrary lengths can be roughly seen as a flexible object type in programming languages that can resemble any data structure. Data structure operations can then be defined as abstract functions and appropriate theory to express the properties. We will see how we state properties about these types. We have also used `Bits(*)` for tagged keys. Although tagged keys can be encoded as tuples of raw keys (`Bits(*)`) and an algorithm tag (`Maybe(Integer)` because DH and 0salt keys have null algorithm tags), we have chosen to hide the structure of key behind bitstring. This requires using abstract functions to extract the algorithm tag but does not proliferate packages and SMT-LIB

code with unnecessary details.

Observe that the stateless package `Names` is used to avoid hardcoded magic integers for the key names all around the project. Moreover, it provides other oracles such as `PARENTS` that implements the function PrntN from the Section 3.2 and return the parent key names of the given key name. Similarly, the stateless package `Labels` expose one oracle `oracle LABEL(n: Integer, r: Bool) -> Integer` that implements the function Labels. Unlike the security model in the Section 3.2 where $r \in \{\texttt{res}, \texttt{ext}, \bot\}$, r is boolean such that $r =$ **false** corresponds to `ext` and $r =$ **true** corresponds to `res`. The oracle `LABEL` ignores r for $n \neq bind$ as it does not affect those key computations.

We have separated the state of the `Map` package (i.e. the mapping table) into a separate package `MapTable` exposing getters and setters to access the mapping table without additional logic. This is very useful for translating games $\texttt{Gks}^{\text{MapXpd}}$ and $\texttt{Gks}^{1}(\mathcal{S})$ while reusing the code of package `Map`. Recall that the the only difference of the packages `Map`, `MapXpd`, and `MapXpdRemap` was the code of oracles $\mathsf{XPD}_{n \in O^*, \ell}$ which derived the output keys. Therefore, we only translate the new code of oracles $\mathsf{XPD}_{n \in O^*, \ell}$ in packages `MapXpd` and `MapXpdRemap` into packages `MapXpd` and `MapXpdRemap` in SSBee while forwarding other calls to the package `Map`. As a result, the mapping table needs to be shares between `Map` and `MapXpd` or `MapXpdRemap`. Although we do not verify Lemma C.5 of [BDLE+21] in this thesis, we have translated the games $\texttt{Gks}^{\text{MapXpd}}$ and $\texttt{Gks}^{1}(\mathcal{S})$ in our SSBee project. Thus, we refer the reader to the code repository [Raj25b] and encourage them to check the diagrams for compositions `GksMapXpd` and `Gks1`.

4.1.2 Key package

We now explain the code of collapsed packages `Key` and `Log`. Listing 8 shows the code of package `Key` in SSBee.

```
1  package Key {
2      params {
3          handle_alg: fn Bits(*) -> Integer, /* returns the algorithm identifier of the
       given handle */
4          key_alg: fn Bits(*) -> Integer, /* returns the algorithm identifier of the given
       key */
5          tag: fn Integer, Bits(*) -> Bits(*), /* tags the given key with the given
       algorithm identifier */
6          untag: fn Bits(*) -> Bits(*), /* untags the given key and returns a raw key */
7          name: fn Bits(*) -> Integer, /* returns the name of the given handle */
8          level: fn Bits(*) -> Maybe(Integer), /* returns the level of the given handle */
9          zeros: fn Integer -> Bits(*), /* returns an all zeros bitstring of the given
       length */
10         len_key: fn Bits(*) -> Integer /* returns the length of the given key */
11     }
12
13     state {
14         K: Table((Integer, Integer, Bits(*)), (Bits(*), Bool)) /* maps (name, level,
       handle) to (key, honesty bit) */
15     }
16
```

```
    import oracles {
        GET_KEY_PACKAGE_IDEALIZATION_PARAMETER(n: Integer, l: Integer) -> Bool,
        SAMPLE(n: Integer) -> Bits(*),
        UNQ(n: Integer, h: Bits(*), hon: Bool, k: Bits(*)) -> Bits(*),
        Q(n: Integer, h: Bits(*)) -> Maybe(Bits(*)),
        IS_DH_KEY(n: Integer) -> Bool,
        IS_PSK(n: Integer) -> Bool,
        IS_0salt(n: Integer) -> Bool,
        IS_0ikm(n: Integer) -> Bool,
        IS_0salt_HANDLE(h: Bits(*)) -> Bool,
        IS_0ikm_HANDLE(h: Bits(*)) -> Bool,
        IS_noPSK_HANDLE(h: Bits(*)) -> Bool,
        IS_noDH_HANDLE(h: Bits(*)) -> Bool,
        IS_HASH_ALGORITHM_SUPPORTED(alg: Integer) -> Bool,
        GET_HASH_ALGORITHM_LENGTH(alg: Integer) -> Integer,
    }

    oracle SET(n: Integer, l: Integer, h: Bits(*), hon: Bool, ks: Bits(*)) -> Bits(*) {
        assert (name(h) == n);
        is_dh_key <- invoke IS_DH_KEY(n);
        if not is_dh_key {
            assert (Unwrap(level(h)) == l);
            h_alg <- handle_alg(h);
            is_hash_alg_supported <- invoke IS_HASH_ALGORITHM_SUPPORTED(h_alg);
            assert is_hash_alg_supported;
            assert (key_alg(ks) == h_alg);
            k <- untag(ks);
            len_h <- invoke GET_HASH_ALGORITHM_LENGTH(h_alg);
            assert (len_h == len_key(k));
            b <- invoke GET_KEY_PACKAGE_IDEALIZATION_PARAMETER(n, l);
            if b {
                if hon {
                    k <- invoke SAMPLE(len_h);
                }
            }
        } else {
            assert (level(h) == None);
            k <- ks;
        }
        q_h <- invoke Q(n, h);
        if (q_h != None as Bits(*)) {
            return Unwrap(q_h);
        }
        unq_h <- invoke UNQ(n, h, hon, k);
        if (h != unq_h) {
            return unq_h;
        }
        if is_dh_key {
            K[(n, 0, h)] <- Some((k, hon));
        } else {
            K[(n, l, h)] <- Some((k, hon));
        }
        return h;
    }
```

```
71
72      oracle GET(n: Integer, l: Integer, h: Bits(*)) -> (Bits(*), Bool) {
73          h_alg <- handle_alg(h);
74          len_h <- invoke GET_HASH_ALGORITHM_LENGTH(h_alg);
75          is_dh_key <- invoke IS_DH_KEY(n);
76          if is_dh_key {
77              is_noDH_handle <- invoke IS_noDH_HANDLE(h);
78              if is_noDH_handle {
79                  return (tag(h_alg, zeros(len_h)), false);
80              }
81              assert (K[(n, 0, h)] != None as (Bits(*), Bool));
82              (ks, hon) <- parse Unwrap(K[(n, 0, h)]);
83              k <- tag(h_alg, ks);
84              return (k, hon);
85          }
86          is_0salt <- invoke IS_0salt(n);
87          if is_0salt {
88              is_0salt_handle <- invoke IS_0salt_HANDLE(h);
89              assert is_0salt_handle;
90              ks <- zeros(1);
91              k <- tag(h_alg, ks);
92              return (k, false);
93          }
94          is_0ikm <- invoke IS_0ikm(n);
95          if is_0ikm {
96              is_0ikm_handle <- invoke IS_0ikm_HANDLE(h);
97              assert is_0ikm_handle;
98              ks <- zeros(len_h);
99              k <- tag(h_alg, ks);
100             return (k, false);
101         }
102         assert (K[(n, l, h)] != None as (Bits(*), Bool));
103         (ks, hon) <- parse Unwrap(K[(n, l, h)]);
104         k <- tag(h_alg, ks);
105         return (k, hon);
106     }
107 }
```

Listing 8: Package `Key`

Notice that packages $\texttt{NKey}_{n\in\{dh,0salt,0ikm\}}$ and $\texttt{Key}^{b}_{n,\ell}$ all are collapsed to this package. Observe that non-leveled key tables K_{dh} storing DH secrets are stored in level zero. Recall that DH handles do not have levels and we also consider their levels to be null. However, storing DH secrets in level zero does not introduce undesired effects as the level is hidden by the adversary. That is, in both oracle `SET` and `GET`, we set and retrieve DH secrets from level zero and the given level argument is ignored. We use abstract functions to tag and untag keys as well as retrieving algorithm tag of a tagged key (`key_alg`). The abstract function `zeroes` returns an all zero string of the given length. One can use a similar approach to use string literals in one's code, i.e. using the output value of an abstract function on specific input(s). An important difference of this implementation with the pseudocode of the package $\texttt{Key}^{b}_{n,\ell}$ is the new assertion in line 41 about whether the hash algorithm specified in the handle is supported. The

assertion should not fail when a key is set by the key derivation function oracles as the algorithm tag of the new handle and key is inherited from the parents. However, the assertion may fail when the adversary sets an honest or dishonest application PSK with an unsupported hash algorithm. BDEFKK delay this abort to the point when the hash algorithm tag is checked by the agile functions xtr and xpd to derive a new key using hmac_{alg}. In such a situation, the game abort is implicit because simply the underlying hash function hash-*alg* called by hmac_{alg} is undefined. We make this assertion explicit and bring it upfront to the key package. As a result, it is ensured that all keys are tagged with a supported hash function and the agile functions xtr and xpd are abort-free and can be viewed as mathematical functions. This assertion is also useful when the a new key is sampled by invocation of oracle `SAMPLE` of the package `Sample`. Listing 9 shows the code of the package.

```
package Sample {
    params {
        cast256: fn Bits(256) -> Bits(*),
        cast384: fn Bits(384) -> Bits(*),
        cast512: fn Bits(512) -> Bits(*),
        default: fn Integer -> Bits(*),
    }

    oracle SAMPLE(n: Integer) -> Bits(*) {
        if (n == 256) {
            k256 <-$ Bits(256);
            return cast256(k256);
        }
        if (n == 384) {
            k384 <-$ Bits(384);
            return cast384(k384);
        }
        if (n == 512) {
            k512 <-$ Bits(512);
            return cast512(k512);
        }
        abort;
        /* This should never happen */
        return default(0);
    }
}
```

Listing 9: Package `Sample`

Since SSBee does not support type casting, we use an abstract function to convert a value of type `Bits(256)` to `Bits(*)`. Notice how the abstract function `default(0)` is used to return a fixed literal for the default case. Due to the check for a supported hash algorithm before sampling, we expect that the default case does not happen.

As a final note, the collapsed package `Key` should cover the semantics of the package $\mathtt{NKey}_n$. Observe that all-zero values are returned for the keys *0salt* and *0ikm* as well as for the handles noPSK and noDH. However, the oracle `SET` does not abort when an all-zero DH secret is about to be set. In order to reduce the complexities of this package, we move this assertion to the DH package. This does not change the semantics for

the code equivalence but is not enough when keys are set directly by the adversary in the package $\mathtt{NKey}_{dh}$. Looking back to Section 3.3, SET_{dh} oracle is only exposed to the adversary in the modular assumption security games $\mathtt{Gxtr2}^{b}_{hs,\ell}$. Nevertheless, one can simply move the assertion to the key package in a code equivalence game hop before reducing to the assumption.

4.1.3 Log package

We now explain how the packages $\mathtt{Log}^{P,map}_{n}$ are translated to SSBee. Listing 10 shows the code of the package `Log`.

```
package Log {
    params {
        /* pattern: Integer, /* pattern see README (shortly; Z: 0, A: 1, D: 2, F: 3) */
        /* mapping: Integer, /* mapping see README (shortly; 0: 0, 1: 1, inf: 2) */
        level: fn Bits(*) -> Maybe(Integer) /* returns the level of the given handle */
    }

    state {
        Log: Table((Integer, Bits(*)), (Bits(*), Bool, Bits(*))), /* maps (name, handle) to (mapped handle, honesty bit, key) */
        Seen: Table((Integer, Bits(*)), Bool), /* indicates whether the (name, key) was assigned before */
        LogInverseDishonest: Table((Integer, Bits(*)), Bits(*)), /* maps (name, key) to first dishonest handle */
        LogInverseDishonestLevelZero: Table(Bits(*), Bits(*)), /* maps psk key to first dishonest handle in level zero */
        LogInverseDishonestLevelNonZero: Table(Bits(*), Bits(*)), /* maps psk key to first dishonest handle in nonzero level */
        J: Table(Bits(*), Bool) /* indicates whether the key was mapped before */
    }

    import oracles {
        GET_LOG_PACKAGE_PARAMETERS(n: Integer) -> (Integer, Integer),
        IS_INFINITY_MAPPING(mapping: Integer) -> Bool,
        IS_1_MAPPING(mapping: Integer) -> Bool,
        IS_A_PATTERN(pattern: Integer) -> Bool,
        IS_D_PATTERN(pattern: Integer) -> Bool,
        IS_F_PATTERN(pattern: Integer) -> Bool,
        IS_PSK(n: Integer) -> Bool,
    }

    oracle UNQ(n: Integer, h: Bits(*), hon: Bool, k: Bits(*)) -> Bits(*) {
        parameters <- invoke GET_LOG_PACKAGE_PARAMETERS(n);
        (pattern, mapping) <- parse parameters;

        /* mapping */
        is_infinity <- invoke IS_INFINITY_MAPPING(mapping);
        if is_infinity {
            if not hon {
                if (LogInverseDishonest[(n, k)] != None as Bits(*)) {
                    hp <- Unwrap(LogInverseDishonest[(n, k)]);
                    Log[(n, h)] <- Some((hp, hon, k));
```

```
                    return hp;
                }
            }
        }

        is_1_mapping <- invoke IS_1_MAPPING(mapping);
        if is_1_mapping {
            if not hon {
                if (J[k] != Some(true)) {
                    r <- Unwrap(level(h));
                    if (r == 0) {
                        if (LogInverseDishonestLevelNonZero[k] != None as Bits(*)) {
                            hp <- Unwrap(LogInverseDishonestLevelNonZero[k]);
                            LogInverseDishonestLevelZero[k] <- Some(h);
                            Log[(n, h)] <- Some((hp, hon, k));
                            J[k] <- Some(true);
                            return hp;
                        }
                    } else {
                        if (LogInverseDishonestLevelZero[k] != None as Bits(*)) {
                            hp <- Unwrap(LogInverseDishonestLevelZero[k]);
                            LogInverseDishonestLevelNonZero[k] <- Some(h);
                            Log[(n, h)] <- Some((hp, hon, k));
                            J[k] <- Some(true);
                            return hp;
                        }
                    }
                }
            }
        }

        /* pattern */
        is_A_pattern <- invoke IS_A_PATTERN(pattern);
        if is_A_pattern {
            r <- Unwrap(level(h));
            if ((r == 0) and not hon) {
                if (LogInverseDishonestLevelZero[k] != None as Bits(*)) {
                    abort;
                }
            }
        }

        is_D_pattern <- invoke IS_D_PATTERN(pattern);
        if is_D_pattern {
            if not hon {
                if (LogInverseDishonest[(n, k)] != None as Bits(*)) {
                    abort;
                }
            }
        }

        is_F_pattern <- invoke IS_F_PATTERN(pattern);
        if is_F_pattern {
            if (Seen[(n, k)] != None as Bool) {
```

```
                abort;
            }
        }

        Log[(n, h)] <- Some((h, hon, k));
        Seen[(n, k)] <- Some(true);
        if not hon {
            LogInverseDishonest[(n, k)] <- Some(h);
            is_psk <- invoke IS_PSK(n);
            if is_psk {
                r <- Unwrap(level(h));
                if (r == 0) {
                    LogInverseDishonestLevelZero[k] <- Some(h);
                } else {
                    LogInverseDishonestLevelNonZero[k] <- Some(h);
                }
            }
        }
        return h;
    }

    oracle Q(n: Integer, h: Bits(*)) -> Maybe(Bits(*)) {
        if (Log[(n, h)] == None) {
            return None;
        }
        (hp, hon, k) <- parse Unwrap(Log[(n, h)]);
        return Some(hp);
    }
}
```

Listing 10: Package `Log`

Observe how the package `Log` retrieves its parameters from the package `Parameters` via the oracle `GET_LOG_PACKAGE_PARAMETERS(n)`.

The translation of package `Log` is unique because there is no possible direct translation from the pseudocode to the SSBee language. Firstly, it is not possible to use an existential quantifier in the code of packages in SSBee. Secondly, SSBee does not support iteration control flow statements (i.e. loops). The oracle UNQ of the package $\texttt{Log}_n^{P,map}$ searches the table Log_n for an entry with specific conditions. Since SSBee does not support loops, we can not translate the table lookup with a lookup loop. However, we mentioned that the existential quantifier is essentially an epsilon operator in Hilbert's epsilon calculus [AZ24]. This intuition leads to a translation of the table lookup as a function that given a table and a value returns an entry with specific conditions on the value or returns $\perp$ when no such entry exists. We have indeed used such an approach in verification of Lemma 5.20 in Section 5 by replacing the epsilon operator with an abstract function. However, there is an inherent nondeterminisim for the value returned by an epsilon operator while an abstract function is a deterministic mathematical function. If multiple entries exist in the table that satisfy the condition, an epsilon operator may return any of them but an abstract function shall return one of them deterministically. If the table lookup imposes additional properties on the order of search, then this order needs to be taken into account for the output of the abstract

function. Fortunately, order is not important in the table lookups of the $\texttt{Log}_n^{P,map}$ package and any entry can be chosen with the mentioned properties. Although we believe this translation approach creates more resemblance with the pseudocode of $\texttt{Log}_n^{P,map}$ package and possibly and easier translation, we have chosen another approach for the translation for this case. The first reason is the timing. We discovered the epsilon operator approach can be useful for lookup table translation only after we had generated the code of package `Log` in SSBee and proved properties about it. The second reason is that we speculate it might be more heavy for the SMT solver as we need existential quantifiers for the case when multiple entries exist. We have experienced that universal and existential quantifiers have significantly contributed to the unknown responses we get from the SMT solver. Thus, we leave translation of the $\texttt{Log}_n^{P,map}$ package with an epsilon operator implemented as an abstract function to a future work.

The other approach is to use *inverse* tables. This approach only works for the table lookups that given k, we are searching for an index h in the table T such that $T[h] = k$ or $T[h] = (k, \dots)$. The idea is to store another table T^{-1} that maps values k to indices h. Since the table T is not necessarily an injective mapping, T^{-1} is not an inverse mapping. Depending on how T^{-1} is set, we have different guarantees about the index $h = T^{-1}[k]$. Nevertheless, we abuse the notation of inverse mapping to emphasize the swap of range and domain more importantly for the following property: for all k, if $T^{-1}[k] \neq \bot$, then $T[T^{-1}[k]] = (k, \dots)$. One might also need that if $T^{-1}[k] = \bot$, then there does not exist an index h such that $T[h]$ contains k.

Concretely, in the package `Log`, we perform table lookups for all pattern parameters except $P = Z$ and all mapping parameters except $map = 0$. For the infinity mapping ($map = \infty$), given a handle h, honesty bit hon, and key k, if h is dishonest ($hon =$ **false**), we search the table Log for any handle h' such that $\text{Log}[h'] = (h', \textbf{false}, k)$. Namely, (1) h' is mapped to handle h', (2) h' is dishonest, and (3) h' is mapped to key k. From Figure 18, recall that the mapping condition is $P_{map} = hon = hon' =$ **false**; however, we can check $hon =$ **false** upfront (line 34) before the table lookup as it does not concern the table. We introduce the table `LogInverseDishonest` that maps a key k to handle h' if $\text{Log}[h'] = (h', \textbf{false}, k)$. We indeed prove the expected property `Log[ LogInverseDishonest[k]] = (LogInverseDishonest[k], false, k)` as an invariant state relation that is preserved across all queries. To achieve this property, we update the table `LogInverseDishonest` whenever we update the table `Log` with required conditions. Table `Log` is set at two points: (1) at line 37 when an infinity mapping triggers and (2) at line 96 when no mapping or pattern triggers. Notice that in the first case table `Log` is not set with an entry of the form $\text{Log}[h] = (h, \_, \_)$, i.e. h is not mapped to itself. That is because oracle `UNQ` is called by the oracle `SET` of the package `Key` which checks whether the handle h exists in the Log table or not via the oracle call `Q(h)`. Therefore, if `hp = h` in these two cases, due to the property `Log[LogInverseDishonest[k]] = (LogInverseDishonest [k], false, k)`, handle `h` should already exist in the table, which is a contradiction. However, at line 96, table `Log` is always set with $\text{Log}[h] = (h, \_, k)$ for the given k. As a result, it only remains to check $hon =$ **false** at line 98 and and only then the table `LogInverseDishonest` can be updated at line 99. Finally, at line 35, we lookup the Log table by checking whether `LogInverseDishonest[k]` is not none. Since all packages $\texttt{Log}_n^{P,map}$ for all key names n are collapsed to one package `Log`, indices of all tables also

include the key name. Since the D pattern also has the same table lookup condition, we can reuse the table `LogInverseDishonest`.

The same idea is used to for the mapping $map = 1$ and pattern $P = A$. Recall from Figure 18 that for $map = 1$, given a handle h, honesty bit hon, and key k, if $hon = \textbf{false}$, we search the Log table for an index h' such that (1) $\text{Log}[h'] = (h', \textbf{false}, k)$, (2) $\{\mathsf{level}(h), \mathsf{level}(h')\} = \{0, \ell\}$ for $\ell \neq 0$, and (3) $J[k] = \textbf{false}$, i.e. h' is a dishonest handle mapped to h' and key k and exactly one of h and h' are in level zero. In other words, if $\mathsf{level}(h) = 0$, we look for an index handle h' in a higher level but if $\mathsf{level}(h) > 0$, we look for an index in level zero. Translating to SSBee, we first check $hon = \textbf{false}$ at line 45 and ensure $J[k] \neq \textbf{true}$ at line 46. The reason for checking $J[k] \neq \textbf{true}$ instead of $J[k] = \textbf{false}$ is that table entries can be either null or contain some value. Moreover, all table entries are null in the beginning. Since we never set false in the table, there are essentially two possible values for $J[k]$: `None` or `Some(true)`. If we wanted to check $J[k] = \textbf{false}$, we would need to initialize the table with **false**. Finally, we check the level r of the handle h and, based on r, lookup in the tables `LogInverseDishonestLevelNonZero` or `LogInverseDishonestLevelZero` respectively for a dishonest handle h' with non zero level $r' > 0$ or a dishonest handle h' at level zero. Formally, these tables map key k to handle h' if $\text{Log}[h'] = (h'', \textbf{false}, k)$ and $\mathsf{level}(h') = 0$ or $\mathsf{level}(h') > 0$. Notice that h'' is not necessary the same as h and this is intended. We will shortly explain why this is the case. We again prove the following expected properties: `Log[LogInverseDishonestLevelZero[k]] = (_, false, k)` and `Log[LogInverseDishonestLevelNonZero[k]] = (_, false, k)` as an invariant state relation. We also show that `level(LogInverseDishonestLevelZero[k]) = 0` while `level(LogInverseDishonestLevelNonZero[k]) != 0`. Accordingly, we have to update these two tables when we update the Log table at lines 52, 60, and 96. Similar to the infinity mapping case, these tables are updated in lines 104 and 106 depending on the level of h. Unlike the infinity mapping, these tables are also updated when a mapping occurs. Similar to the infinity mapping, it can be even argued that `hp != h` and intuitively we should not update the tables. Looking back, we do not expect the tables `LogInverseDishonestLevelNonZero` or `LogInverseDishonestLevelZero` to map k to handle h' such that $\text{Log}[h'] = (h'', \textbf{false}, k)$ where $h'' = h'$ but rather the honesty bit, key, and level of h' are important. Looking ahead, this property is very useful when proving invariance of the following state relation:

$$\text{LogInverseDishonestLevelZero}[k] \neq \bot \wedge \text{LogInverseDishonestLevelNonZero}[k] \neq \bot \implies J[k] = \textbf{true} \qquad \text{(Invariant-J)}$$

In other words, $J[k]$ is an indictor for whether a mapping has occurred. One can also view this choice of definition for the tables `LogInverseDishonestLevelNonZero` and `LogInverseDishonestLevelZero` in the context of the security model. The mapping pattern $map = 1$ is used to map the first time a dishonest resumption PSK collides with a dishonest application PSK. Hence, before a mapping of key k happens, there is no collision between dishonest application and resumption PSKs. That is, for every k not involved in a mapping, either there exists no handle h' with level zero such that $\text{Log}[h'] = (h', \textbf{false}, k)$ or there exists no handle h' with nonzero level such that

$\mathrm{Log}[h'] = (h', \mathbf{false}, k)$. We indeed prove the following as another invariant state relation:

$$\texttt{LogInverseDishonestLevelZero}[k] = \bot \wedge \texttt{LogInverseDishonestLevelNonZero}[k] \neq \bot$$
$$\implies \texttt{LogInverseDishonestLevelNonZero}[k] = \texttt{LogInverseDishonest}[k]$$
$$\texttt{LogInverseDishonestLevelZero}[k] \neq \bot \wedge \texttt{LogInverseDishonestLevelNonZero}[k] = \bot$$
$$\implies \texttt{LogInverseDishonestLevelZero}[k] = \texttt{LogInverseDishonest}[k]$$

Together with `Log[LogInverseDishonest[k]] = (LogInverseDishonest[k], false, k)`, one can conclude:

$$\texttt{LogInverseDishonestLevelZero}[k] = \bot \wedge \texttt{LogInverseDishonestLevelNonZero}[k] \neq \bot$$
$$\implies \texttt{Log}[\texttt{LogInverseDishonestLevelNonZero}[k]] = (\texttt{LogInverseDishonestLevelNonZero}[k], \mathbf{false}, k)$$
$$\texttt{LogInverseDishonestLevelZero}[k] \neq \bot \wedge \texttt{LogInverseDishonestLevelNonZero}[k] = \bot$$
$$\implies \texttt{Log}[\texttt{LogInverseDishonestLevelZero}[k]] = (\texttt{LogInverseDishonestLevelZero}[k], \mathbf{false}, k)$$

Based on the constrapositive of Invariant-J, one of the entries `LogInverseDishonestLevelNonZero[k]` or `LogInverseDishonestLevelZero[k]` are null before a mapping (i.e. $J[k] = \mathbf{false}$). However, the moment a mapping occurs for a key k, say without loss of generality $\mathrm{Log}[h] = (h', \mathbf{false}, k)$ where $\mathsf{level}(h) = 0$ but $\mathsf{level}(h') > 0$, there is no handle h in the table with level $\mathsf{level}(h) = 0$ such that $\mathrm{Log}[h] = (h, \mathbf{false}, k)$. Clearly, future queries of the adversary with the same key k might create such an entry as the mapping happens only once.

In Section 3.2, we mentioned that the Key and Log tables need to be initialized for the handles $\mathsf{noPSK}\langle alg \rangle$. We initialize these tables in the SMT-LIB code as a lemma. Moreover, we need to initialize the inverse table for the all-zero keys $0^{\mathsf{len}(alg)}$ to map to the $\mathsf{noPSK}\langle alg \rangle$ handles.

Notice the the table `LogInverseDishonestLevelZero[k]` can be reused for the pattern A which searches the table for a collision between level zero dishonest keys.

Finally, we introduce the table `Seen` for the use of pattern F. This table records whether a key has ever been set in the table as the pattern F aborts whenever the same key is about to be set again.

4.1.4 Other packages

We will see the code of some of other packages in the next section but refer the reader to the repository for the code of all other packages, such as `Xtr`, `Xpd`, `Map`, `MapTable`, `DH`, `Hash`, and `Check`. All these packages except `Hash` have a direct translation from their pseudocode to SSBee language, although they extensively use abstract functions for various operations out of the scope of the model. For example, the package `DH` uses an abstract function `exp: fn Bits(*), Integer -> Bits(*)` to raise DH share (`Y: Bits(*)`) to the power of private exponent (`x: Integer`) and return the share Y^x. Interestingly, we do not even need to state the commutative property of the exponentiation (i.e. $(g^x)^y = (g^y)^x$) for the code equivalence proof obligations. We decide not to proliferate the thesis with the lengthy abstract function declarations.

The only exception to direct translation is the package `Hash`. The oracle `HASH(t)` exposed by the ideal package $\texttt{Hash}^1$ performs a table lookup to ensure no other transcript t' hashes to the same digest of transcript t. As a result, this captures the collision resistance of the underlying hash function. We have translated this package with the exactly same technique of an inverse table mapping hash digests to the transcripts. Since we only focus at the code equivalence of Lemma C.2 of [BDLE+21] and the package `Hash` is not idealized in the games `Gks0` and `Gks0Map`, we refrain from further discussion and refer the reader to the repository. Moreover, the oracle `HASH(t)` is a total function and does not abort in the real package $\texttt{Hash}^0$. Thus, we replace the call to `HASH(t)` with an abstract function `hash1(t)`. However, it is noteworthy that this is not a sound translation when verifying Lemma C.5 of [BDLE+21] where the package $\texttt{Hash}^0$ is idealized and the oracle `HASH(t)` can abort.

4.2 Towards verification of Lemma C.2

We now proceed to the code equivalence proof in SSBee. Listing 11 shows the high level structure of the proof file `proof-lemma-c2.ssp`.

```
proof LemmaC2 {
    ...
    instance game_Gks0 = Gks0 {
        params {
            ...
        }
    }
    instance game_Gks0Map = Gks0Map {
        params {
            ...
        }
    }
    gamehops {
        equivalence game_Gks0 game_Gks0Map {
            DHGEN: {
                invariant: [
                    ./proofs/abstract-functions.smt2
                    ./proofs/invariants.smt2
                    ./proofs/oracles/DHGEN.smt2
                ]

                lemmas {
                    invariant: [no-abort]
                    same-output: [no-abort]
                    equal-aborts: []
                }
            }
            DHEXP: {
                invariant: [
                    ./proofs/abstract-functions.smt2
                    ./proofs/randomness-mapping.smt2
                    ./proofs/invariants.smt2
                    ./proofs/oracles/DHEXP.smt2
```

```
            ]

            lemmas {
                same-output: [no-abort]
                equal-aborts: []
                invariant: [no-abort]
            }
        }
        SET: {
            invariant: [
                ./proofs/abstract-functions.smt2
                ./proofs/randomness-mapping.smt2
                ./proofs/invariants.smt2
                ./proofs/oracles/SET.smt2
            ]

            lemmas {
                all-invariants-after: [no-abort, all-invariants-before, lemma-rand-is
-eq]
                invariant: [no-abort, lemma-rand-is-eq]
                same-output: [no-abort]
                equal-aborts: []
            }
        }
        GET: {
            invariant: [
                ./proofs/abstract-functions.smt2
                ./proofs/randomness-mapping.smt2
                ./proofs/invariants.smt2
                ./proofs/oracles/GET.smt2
            ]

            lemmas {
                lemma-left-output: [no-abort, lemma-name-and-level-of-handle]
                lemma-right-output: [no-abort, lemma-name-and-level-of-handle]
                same-output: [no-abort, lemma-left-output, lemma-right-output, lemma-
name-and-level-of-handle, lemma-alg-is-preserved]
            }
        }
        XTR: {
            invariant: [
                ./proofs/abstract-functions.smt2
                ./proofs/randomness-mapping.smt2
                ./proofs/invariants.smt2
                ./proofs/oracles/XTR.smt2
            ]

            lemmas {
                same-output: [no-abort, lemma-Gks0-output, lemma-Gks0Map-output]
                lemma-Gks0Map-output: [no-abort]
                lemma-Gks0-output: [no-abort]
            }
        }
        XPD: {
```

```
                invariant: [
                    ./proofs/abstract-functions.smt2
                    ./proofs/randomness-mapping.smt2
                    ./proofs/invariants.smt2
                    ./proofs/oracles/XPD.smt2
                ]

                lemmas {
                    same-output: [no-abort, lemma-Gks0-output, lemma-Gks0Map-output]
                    lemma-Gks0Map-output: [no-abort]
                    lemma-Gks0-output: [no-abort]
                }
            }
        }
    }
}
```

Listing 11: Proof file `proof-lemma-c2.ssp`

As we saw in Section 2.5, SSBee relies on Theorem 2.3 and asks the user to provide state relations and try to prove their invariance as well as same-output and equal-aborts properties for each of the oracles exposed to the adversary. Games `Gks0` and `Gks0Map` expose five oracles `DHGEN`, `DHEXP`, `GET`, `SET`, `XTR`, and `XPD` to the adversary. We have managed to point out all necessary state relations to prove the same-output property of all oracles. Moreover, for the oracles `DHGEN`, `DHGEN`, and `SET`, we also prove the equal-aborts property as well as invariance of state relations required for their proof. This leaves proving invariance of other state relations and the equal-aborts property for the other three oracles to a future work.

Although SSBee uses the keyword `invariant` for referring to the list of related SMT-LIB files, it internally concatenates all of them and include them in its compiled SMT-LIB output file. Benefitting from this, we have organized our SMT-LIB files as follows: `abstract-functions.smt2` include the definitions of abstract function `log_package_parameters` described before and key parents function `parents(n)`; `randomness-mapping.smt2` include the template of all randomness mapping definitions shared by all oracles; `invariants.smt2` includes all state relations for the equivalence; `{ORACLE}.smt2` includes lemmata and invariants necessary for proof obligations of each oracle. We now briefly explain the verification process for some oracles.

4.2.1 Oracle DHEXP

The following listings show the code of oracle `DHEXP` exposed by the packages `DH` and `Map`.

```
1  package DH {
2      params {
3          exp: fn Bits(*), Integer -> Bits
       (*),
4          mk_dh_handle: fn Bits(*), Bits(*)
        -> Bits(*),
5          ...
6      }
7      state {
8          E: Table(Bits(*), Integer) /*
       maps g^x to x */
9      }
10     import oracles {
11         SET(n: Integer, l: Integer, h:
       Bits(*), hon: Bool, ks: Bits(*)) ->
       Bits(*),
12         GET_DH_NAME() -> Integer,
13     }
14     ...
15     oracle DHEXP(X: Bits(*), Y: Bits(*))
       -> Bits(*) {
16         assert (grp(X) == grp(Y));
17         h <- mk_dh_handle(X, Y);
18         hon_X <- not (E[X] == None);
19         hon_Y <- not (E[Y] == None);
20         assert (hon_X == true);
21         x <- Unwrap(E[X]);
22         k <- exp(Y, x);
23         if (k == zeros(len_key(k))) {
24             abort;
25         }
26         hon <- (hon_X and hon_Y);
27         dh <- invoke GET_DH_NAME();
28         h <- invoke SET(dh, 0, h, hon,
       encode_group_member(k));
29         return h;
30     }
31     ...
32 }
```

```
1  package Map {
2      params {
3          mk_dh_handle: fn Bits(*), Bits(*)
        -> Bits(*),
4          ...
5      }
6      import oracles {
7          GET_DH_NAME() -> Integer,
8          DHEXP(X: Bits(*), Y: Bits(*)) ->
       Bits(*),
9          SETMAP(n: Integer, l: Integer,
       ext_h: Bits(*), int_h: Bits(*)),
10         ...
11     }
12     ...
13     oracle DHEXP(X: Bits(*), Y: Bits(*))
       -> Bits(*) {
14         h <- mk_dh_handle(X, Y);
15         int_h <- invoke DHEXP(X, Y);
16         dh <- invoke GET_DH_NAME();
17         _ <- invoke SETMAP(dh, 0, h,
       int_h);
18         return h;
19     }
20     ...
21 }
```

We usually start with proving the same-output property and then proceed to equal-aborts and finally prove the invariance of state relations. SSBee tries to prove the left and right `DHEXP` oracles return the same output given (1) the same inputs are passed to the oracles (i.e. DH shares X and Y), (2) state relations hold on the states of the games before the oracle query, (3) oracles consume the randomness strings, and (4) oracles do not abort. Notice that the only probabilistic oracles are SET and DHGEN. We can simply define a randomness mapping between the sampling points in these oracles similar to the randomness mapping of Section 2.5.

We first try to prove the same-output property. Since we assume the oracles do not abort, the left oracle returns at line 29 while the right oracle returns at line 18. The right oracle returns the the handle constructed by the abstract function `mk_dh_handle(X, Y)`. SSBee can also verify this independently when expressed as a standalone lemma

`lemma-Gks0Map-output`:

```
DHEXP: {
    invariant: [
        ...
    ]
    lemmas {
        lemma-Gks0Map-output: [no-abort]
        lemma-Gks0-output: [no-abort]
        ...
    }
}
```

```
(define-fun <relation-lemma-Gks0Map-output-game_Gks0-game_Gks0Map-DHEXP>
    (
        (old-state-Gks0 <GameState_Gks0_<$$>>)
        (old-state-Gks0Map <GameState_Gks0Map_<$$>>)
        (return-DHEXP-Gks0 <OracleReturn-Gks0-<$$>-DH-<$$>-DHEXP>)
        (return-DHEXP-Gks0Map <OracleReturn-Gks0Map-<$$>-Map-<$$>-DHEXP>)
        (X Bits_*)
        (Y Bits_*)
    )
    Bool
    (=
        (return-value (<oracle-return-Gks0Map-<$$>-Map-<$$>-DHEXP-return-value-or-abort>
    return-DHEXP-Gks0Map))
        (<<func-mk_dh_handle>> X Y)
    )
)
```

However, the left oracle return the handle returned by the oracle SET and the lemma `lemma-Gks0-output` fails to prove (SMT solver reports unknown):

```
(define-fun <relation-lemma-Gks0-output-game_Gks0-game_Gks0Map-DHEXP>
    (
        (old-state-Gks0 <GameState_Gks0_<$$>>)
        (old-state-Gks0Map <GameState_Gks0Map_<$$>>)
        (return-DHEXP-Gks0 <OracleReturn-Gks0-<$$>-DH-<$$>-DHEXP>)
        (return-DHEXP-Gks0Map <OracleReturn-Gks0Map-<$$>-Map-<$$>-DHEXP>)
        (X Bits_*)
        (Y Bits_*)
    )
    Bool
    (=
        (return-value (<oracle-return-Gks0-<$$>-DH-<$$>-DHEXP-return-value-or-abort>
    return-DHEXP-Gks0))
        (<<func-mk_dh_handle>> X Y)
    )
)
```

Observe that the oracle SET call the oracle Q of the package `Log`. The Log package for DH secrets has the parameters $(P, map) = (Z, 0)$. Therefore, no mapping or abort happens and for all DH handles h, if $\text{Log}_{\text{left}}[h] \neq \perp$, then $\text{Log}_{\text{left}}[h] = (h, hon, k)$ for some honesty bit *hon* and key k. BDEFKK refer to this property of the package Log^{Z}_{dh} by *invariant 2a (v)*. Consequently, the same given handle h is returned in all cases by

the oracle SET and the same constructed handle `mk_dh_handle(X, Y)` is retuned to the adversary. Unfortunately, SSBee (or the SMT solver) does not have this information when the oracle Q is called and proving same-output fails with unknown from the SMT solver. We define *invariant 2a (v)* as a state relation as follows and successfully prove the same-output property.

```
(define-fun invariant-2a-v
    (
        (state-Gks0 <GameState_Gks0_<$$>>)
        (state-Gks0Map <GameState_Gks0Map_<$$>>)
    )
    Bool
    ; n = name(h)
    ; Invariant (2a) (v) : Log_left[(n, h)] = some(h, hon, k) or none
    (let
        (
            (Log_left (<pkg-state-Log-<$$>-Log> (<game-Gks0-<$$>-pkgstate-pkg_Log> state-
    Gks0)))
        )
        (forall
            (
                (h Bits_*)
            )
            (let
                (
                    (n (<<func-name>> h))
                )
                (let
                    (
                        (log_entry (select Log_left (mk-tuple2 n h)))
                    )
                    (=>
                        (not ((_ is mk-none) log_entry))
                        (= (el3-1 (maybe-get log_entry)) h)
                    )
                )
            )
        )
    )
)
```

Although this invariant was recognized and expressed by BDEFKK, they overlooked its necessity for proving same-output property of the oracle DHEXP, rendering the importance of the formal verification.

The other proof obligation is equal-aborts. The right oracle queries the DHEXP oracle of the package DH of the game Gks0Map at line 15. Hence, we should consider a similar code to the left oracle is executed at that point. Nevertheless, both oracle assert the shares X and Y belong to the same group (line 12 on left). This assertion passes or fails on both sides due to equality of inputs passed to the oracles. Line 16 on the left asserts the share X exists in the table. A similar assertion occurs when the right oracles calls DHEXP at line 15. In order for the assertions to have the same failure status, we need a state relation for the equality of the exponent tables, i.e. $E_{\text{left}} = E_{\text{right}}$. The

same state relation is necessary to argue the computed keys `k = exp(Y, x)` are the same on the left and right to show the abort command at line 24 of the left oracle has the same failure status as the same command on the right. BDEFKK refers to this state relation by *invariant (1)* and it is expressed as follows in the SMT-LIB language:

```
(define-fun invariant-1
    (
        (state-Gks0 <GameState_Gks0_<$$>>)
        (state-Gks0Map <GameState_Gks0Map_<$$>>)
    )
    Bool
    ; Invariant (1) : exponent table consistency : E_left = E_right
    (let
        (
            (E_left  (<pkg-state-DH-<$$>-E> (<game-Gks0-<$$>-pkgstate-pkg_DH> state-Gks0)
    ))
            (E_right (<pkg-state-DH-<$$>-E> (<game-Gks0Map-<$$>-pkgstate-pkg_DH> state-
    Gks0Map)))
        )
        (= E_left E_right)
    )
)
```

4.2.2 **Oracle** SET

We do not go into every details of the verification process but we highlight the important results and observations from the verification. Firstly, the verification of oracle SET is important because it points out an important difference of the games `Gks0` and `Gks0Map`, i.e., the mapping parameters of the packages $\texttt{Log}^{A}_{psk}$ and $\texttt{Log}^{A1}_{psk}$. The adversary can set honest and dishonest application PSKs through this oracle and it is crucial that the left and right oracles satisfy the equal-aborts property. Essentially, we need to ensure the addition of mapping does not prevent an abort situation in the right game. In other words, it should not be possible that the left game aborts because the given dishonest application PSK handle h collides with another dishonest application PSK h' in the Log table while the right game maps h to some dishonest resumption PSK h''. This is a contradiction because h'' and h' already have a collision (and should have been mapped to each other) but the mapping happens only once. BDEFKK express this property of one-time mapping under *invariant 2e* as follows:

$$\forall k : \left(\begin{array}{l} \exists h \neq h' \wedge \mathsf{level}(h) = 0 \wedge \mathsf{level}(h') \neq 0 \wedge \\ \quad Log^{\text{right}}_{psk}[h] = (\_, 0, k) \\ \quad \wedge \; Log^{\text{right}}_{psk}[h'] = (\_, 0, k)) \end{array}\right\} \Rightarrow J_{psk}[k] = 1 \qquad \text{(Invariant-2e)}$$

We avoid the existential quantifier with our tables `LogInverseDishonestLevelZero` and `LogInverseDishonestLevelNonZero`: (cf. equation Invariant-J)

```
(define-fun J-invariants
    (
```

```
        (old-state-KeyLogGks0Map <GameState_KeyLogGks0Map_<$$>>)
    )
    Bool
    (let
        (
            (J (<pkg-state-Log-<$$>-J> (<game-KeyLogGks0Map-<$$>-pkgstate-pkg_Log> old-state-KeyLogGks0Map)))
            (LogInverseDishonestLevelZero (<pkg-state-Log-<$$>-LogInverseDishonestLevelZero> (<game-KeyLogGks0Map-<$$>-pkgstate-pkg_Log> old-state-KeyLogGks0Map)))
            (LogInverseDishonestLevelNonZero (<pkg-state-Log-<$$>-LogInverseDishonestLevelNonZero> (<game-KeyLogGks0Map-<$$>-pkgstate-pkg_Log> old-state-KeyLogGks0Map)))
        )
        (forall
            (
                (k Bits_*)
            )
            (let
                (
                    (J_k (select J k))
                    (LogInverseDishonestLevelZero_k (select LogInverseDishonestLevelZero k))
                    (LogInverseDishonestLevelNonZero_k (select LogInverseDishonestLevelNonZero k))
                )
                (and
                    ;   J = None or some(True)
                    (or
                        ((_ is mk-none) J_k)
                        (= J_k (mk-some true))
                    )
                    ;   J[k] = None => LogInverseDishonestLevelZero[k] = None or LogInverseDishonestLevelNonZero[k] = None
                    (=>
                        ((_ is mk-none) J_k)
                        (or
                            ((_ is mk-none) LogInverseDishonestLevelZero_k)
                            ((_ is mk-none) LogInverseDishonestLevelNonZero_k)
                        )
                    )
                )
            )
        )
    )
)
```

In Section 4.1.3, we mentioned that one may need another property from an inverse table that if $T^{-1}[k] = \perp$ then no entry with index h exists in the table T such that $T[h]$ contains k. BDEFKK introduce the state relation *invariant (5)* to relate the Log tables

of the left and right games on application PSK handles.

$$
\begin{aligned}
&n = i = dh \vee (i = (psk, 0) \wedge n = psk) \Rightarrow \\
&\quad (Log_n^{\text{left}}[h] \neq \bot) \Leftrightarrow (M_i^{\text{right}}[h] \neq \bot) \Leftrightarrow (Log_n^{\text{right}}[h] \neq \bot) \\
&\quad Log_n^{\text{left}}[h] \neq \bot \Rightarrow \\
&\qquad Log_n^{\text{left}}[h] = (h, hon, k) \wedge \\
&\qquad Log_n^{\text{right}}[h] = (M_i^{\text{right}}[h], hon', k') \wedge \\
&\qquad (hon, k) = (hon', k') \qquad \text{(Invariant-5)}
\end{aligned}
$$

We need a flavour of the inverse table property to relate the tables `LogInverseDishonestLevelZero` of the left and right games. We define this state relation as follows:

$$\text{LogInverseDishonestLevelZero}_{\text{left}}[k] = \bot \Leftrightarrow \text{LogInverseDishonestLevelZero}_{\text{right}}[k] = \bot$$

Notice that this property is not necessary if we prove as another state relation that `LogInverseDishonestLevelZero[k]` is null if no dishonest level zero handle with the given key k exists in the Log table. Then the property follows from Invariant-5.

BDEFKK have presented a detailed pen-and-paper argument for the same-output and equal-aborts properties. We refer the reader to their paper for more details. Moreover, other invariants that are necessary for the verification are expressed in our code repository. We have successfully verified the equal-aborts and same-output properties as well as invariance of the required state relations.

4.2.3 Oracles GET, XPD, and XTR

These were the trickiest oracles for verification. We only managed to verify the same-output property by finding all the necessary invariants. (See the list of lemmata assumed for the same-output proof goal.) We leave the full verification of these three oracles to a future work. However, we want to emphasize that among these three oracles, verification of even the same-output property for oracles `XTR` and `XPD` are mysteriously difficult for the SMT solver, responding with *unsat* only after a few minutes. Ironically, when we split the proof goals to ensure what the output of each oracle should be, the SMT solver takes again a few minutes to prove unsatisfiability. This is not satisfactory specifically for the right oracle (exposed by the Map package) where the output is trivially the constructed handles $\mathsf{xtr}\langle n, h_1, h_2\rangle$ or $\mathsf{xpd}\langle n, label, h_1, args\rangle$. Listing below shows the trivial lemmata `lemma-Gks0Map-output` that takes a long time to verify. We speculate that this phenomena happens due to the complexity of other oracle calls in the body of the oracles XPD and XTR and can be avoided with modular verification techniques. (See Section 4.4) We leave more investigation for the root of this issue and applying modular verification technique as a future work.

```
(define-fun <relation-lemma-Gks0Map-output-game_Gks0-game_Gks0Map-XTR>
    (
        (old-state-Gks0 <GameState_Gks0_<$$>>)
        (old-state-Gks0Map <GameState_Gks0Map_<$$>>)
        (return-XTR-Gks0 <OracleReturn-Gks0-<$$>-Xtr-<$$>-XTR>)
```

```
        (return-XTR-Gks0Map <OracleReturn-Gks0Map-<$$>-Map-<$$>-XTR>)
        (n Int)
        (l Int)
        (h1 Bits_*)
        (h2 Bits_*)
    )
    Bool
    (=
        (return-value (<oracle-return-Gks0Map-<$$>-Map-<$$>-XTR-return-value-or-abort>
    return-XTR-Gks0Map))
        (<<func-mk_xtr_handle>> n h1 h2)
    )
)
(define-fun <relation-lemma-Gks0Map-output-game_Gks0-game_Gks0Map-XPD>
    (
        (old-state-Gks0 <GameState_Gks0_<$$>>)
        (old-state-Gks0Map <GameState_Gks0Map_<$$>>)
        (return-XPD-Gks0 <OracleReturn-Gks0-<$$>-Check-<$$>-XPD>)
        (return-XPD-Gks0Map <OracleReturn-Gks0Map-<$$>-Check-<$$>-XPD>)
        (n Int)
        (l Int)
        (h Bits_*)
        (r Bool)
        (args Bits_*)
    )
    Bool
    (=
        (return-value (<oracle-return-Gks0Map-<$$>-Check-<$$>-XPD-return-value-or-abort>
    return-XPD-Gks0Map))
        (<<func-mk_xpd_handle>> n (<<func-label1>> n r) h args)
    )
)
```

4.2.4 Invariance of state relations

In previous sections we did not discuss proving the invariance of state relations. Proving the invariance of state relations is indeed the most difficult task for verification because it requires proving the preservation of the same state relations for every oracle. In the following, we first present all the invariants introduced by BDEFKK and mention which invariants we were able to prove. Let $\mathsf{idx}(h)$ be equal to $\mathsf{name}(h)$ if $\mathsf{name}(h) \in \{\mathit{0salt}, \mathit{dh}, \mathit{0ikm}\}$ and $(\mathsf{name}(h), \mathsf{level}(h))$, otherwise.

(1) exponent table consistency: $E^{\text{left}} = E^{\text{right}}$

$\forall h$, let $i := \mathsf{idx}(h)$, $n := \mathsf{name}(h)$:

(2a) K and Log: for $(K_i, Log_n) = (K_i^{\text{left}}, Log_n^{\text{left}})$ and $(K_i, Log_n) = (K_i^{\text{right}}, Log_n^{\text{right}})$

(i) $\quad Log_n[h] = \bot \Rightarrow K_i[h] = \bot$

(ii) $\quad Log_n[h] = (h', *, *)$ with $h' \neq h \Rightarrow K_i[h] = \bot$

$$(iii) \quad Log_n[h] = (h', hon, k) \text{ with } h' \neq h \Rightarrow Log_n[h'] = (h', hon, k)$$

$$(iv) \quad Log_n[h] = (h, hon, k) \Leftrightarrow K_i[h] = (k, hon) \neq \bot$$

$$(v) \quad Log_n^{\text{left}}[h] \neq \bot \Rightarrow Log_n^{\text{left}}[h] = (h, *, *)$$

$$(vi) \quad Log_n^{\text{right}}[h], n \notin \{psk, dh\} \neq \bot \Rightarrow Log_n^{\text{right}}[h] = (h, *, *)$$

$$(vii) \quad \forall n' : Log_{n'}[h] \neq \bot \Rightarrow \mathsf{name}(h) = n'$$

$$(viii) \quad Log[h] = (h', *, k) \Rightarrow |k| = \mathsf{len}(h) = \mathsf{len}(h') \wedge \mathsf{alg}(h) = \mathsf{alg}(h')$$

$$(ix) \quad n \in \{0ikm, 0salt, dh\} \Rightarrow (K_n[h] \neq \bot \Rightarrow \mathsf{level}(h) = \bot)$$

$$(x) \quad \forall \ell : n \notin \{0ikm, 0salt, dh\} \Rightarrow (K_{n,\ell}[h] \neq \bot \Rightarrow \mathsf{level}(h) = \ell)$$

(2b) Available keys (mapped-unmapped):

$$M_i^{\text{right}}[h] = h' \neq \bot \Rightarrow Log_n^{\text{right}}[h'] = (h', *, *)$$

(2c) Available keys (right-left):

$$M_i^{\text{right}}[h] = h' \neq \bot \iff Log_n^{\text{left}}[h] = (h, *, *)$$

(2d)

$$\begin{aligned}
&M_i^{\text{right}}[h] \neq \bot \Rightarrow \\
&\quad \mathsf{xtr}\langle n, h_1, h_2\rangle = h \wedge \mathsf{idx}(h_1) = i_1 \wedge \mathsf{idx}(h_2) = i_2 \Rightarrow \\
&\quad\quad M_{i_1}^{\text{right}}[h_1] \neq \bot \wedge M_{i_2}^{\text{right}}[h_2] \neq \bot \wedge \\
&\quad\quad M_i[h] = \mathsf{xtr}\langle n, M_{i_1}^{\text{right}}[h_1], M_{i_2}^{\text{right}}[h_2]\rangle \\
&\quad \mathsf{xpd}\langle n, label, h_1, args\rangle = h \wedge \mathsf{idx}(h_1) = i_1 \wedge n \neq psk \Rightarrow \\
&\quad\quad M_{i_1}^{\text{right}}[h_1] \neq \bot \wedge \\
&\quad\quad M_i^{\text{right}}[h] = \mathsf{xpd}\langle n, label, M_{i_1}^{\text{right}}[h_1], args\rangle \\
&\quad \mathsf{xpd}\langle psk, label, h_1, args\rangle = h \wedge \mathsf{idx}(h_1) = i_1 \Rightarrow \\
&\quad\quad M_{i_1}^{\text{right}}[h_1] \neq \bot \wedge \\
&\quad\quad\quad Log_{psk}[\mathsf{xpd}\langle psk, label, M_{i_1}^{\text{right}}[h_1], args\rangle] = (M_i^{\text{right}}[h], *, *)
\end{aligned}$$

(2e) J-Map:

$$\begin{aligned}
&\forall k : (\exists h \neq h' \wedge \mathsf{level}(h) = 0 \wedge \mathsf{level}((h') \neq 0 \wedge \\
&\left.\begin{array}{l} Log_{psk}^{\text{right}}[h] = (_, 0, k) \\ \wedge\ Log_{psk}^{\text{right}}[h'] = (_, 0, k)) \end{array}\right\} \Rightarrow J_{psk}[k] = 1
\end{aligned}$$

(3) Mapping keeps name and algs: $M_i^{\text{right}}[h] \neq \bot \Rightarrow$
(a) $\mathsf{name}(M_i^{\text{right}}[h]) = \mathsf{name}(h) \wedge$
(b) $\mathsf{alg}(M_i^{\text{right}}[h]) = \mathsf{alg}(h)$.

(4) Children derive their value from their parent(s)
For $Log_n = Log_n^{\text{left}}$ and $Log_n = Log_n^{\text{right}}$

$$
\begin{aligned}
&Log_n[h] \neq \bot \Rightarrow \\
&\quad \mathsf{xtr}\langle n, h_1, h_2\rangle = h \wedge \mathsf{name}(h_1) = n_1 \wedge \mathsf{name}(h_2) = n_2 \Rightarrow \\
&\quad\quad Log_{n_1}[h_1] = (h_1, hon_1, k_1) \neq \bot \wedge \\
&\quad\quad Log_{n_2}[h_2] = (h_2, hon_2, k_2) \neq \bot \wedge \\
&\quad\quad k = \mathsf{xtr}(k_1, k_2) \wedge hon = (hon_1 \vee hon_2) \wedge \\
&\quad\quad Log_n[h] = (*, hon, k) \\
&\quad \mathsf{xpd}\langle n, label, h_1, args\rangle = h \wedge \mathsf{name}(h_1) = n_1 \Rightarrow \\
&\quad\quad Log_{n_1}[h_1] = (h_1, hon_1, k_1) \neq \bot \wedge \\
&\quad\quad k = \mathsf{xpd}(k_1, label, args) \wedge \\
&\quad\quad Log_n[h] = (*, hon_1, k)
\end{aligned}
$$

(5) Consistent logs for input keys:

$$
\begin{aligned}
&n = i = dh \vee (i = (psk, 0) \wedge n = psk) \Rightarrow \\
&\quad (Log_n^{\text{left}}[h] \neq \bot) \Leftrightarrow (M_i^{\text{right}}[h] \neq \bot) \Leftrightarrow (Log_n^{\text{right}}[h] \neq \bot) \\
&\quad Log_n^{\text{left}}[h] \neq \bot \Rightarrow \\
&\quad\quad Log_n^{\text{left}}[h] = (h, hon, k) \wedge \\
&\quad\quad Log_n^{\text{right}}[h] = (M_i^{\text{right}}[h], hon', k') \wedge \\
&\quad\quad (hon, k) = (hon', k')
\end{aligned}
$$

(6) Identical keys and honesty:
$K_i^{\text{left}}[h] = K_{i'}^{\text{right}}[M_i^{\text{right}}[h]]$

We have managed to prove the invariance of state relations (1), (2a), and (2e) across all oracles. We refer the reader to the project repository for the full details of SMT-LIB translation of the relations. We prove invariance of all state relations in separate lemmata in our SMT-LIB code. With this approach, it becomes more clear which invariants are required to prove same-output and equal-aborts properties of an oracle and which invariants are proved for the general soundness of the argument. We use lemmata with names `all-invariants-after` and `all-invariants-before` for asserting the state relations on the game states after and before the oracle call, respectively. For instance the following shows the invariants definition and all-invariants lemmata for the oracle SET.

```
(define-fun <relation-all-invariants-before-game_Gks0-game_Gks0Map-SET>
    (
        (old-state-Gks0 <GameState_Gks0_<$$>>)
        (old-state-Gks0Map <GameState_Gks0Map_<$$>>)
        (return-SET-Gks0 <OracleReturn-Gks0-<$$>-ExternalPskSetter-<$$>-SET>)
```

```
        (return-SET-Gks0Map <OracleReturn-Gks0Map-<$$>-Map-<$$>-SET>)
        (h Bits_*)
        (hon Bool)
        (k Bits_*)
    )
    Bool
    (all-invariants old-state-Gks0 old-state-Gks0Map)
)
(define-fun <relation-all-invariants-after-game_Gks0-game_Gks0Map-SET>
    (
        (old-state-Gks0 <GameState_Gks0_<$$>>)
        (old-state-Gks0Map <GameState_Gks0Map_<$$>>)
        (return-SET-Gks0 <OracleReturn-Gks0-<$$>-ExternalPskSetter-<$$>-SET>)
        (return-SET-Gks0Map <OracleReturn-Gks0Map-<$$>-Map-<$$>-SET>)
        (h Bits_*)
        (hon Bool)
        (k Bits_*)
    )
    Bool
    (all-invariants <<game-state-game_Gks0-new-SET>> <<game-state-game_Gks0Map-new-SET>>)
)
(define-fun invariant
    (
        (state-Gks0 <GameState_Gks0_<$$>>)
        (state-Gks0Map <GameState_Gks0Map_<$$>>)
    )
    Bool
    (and
        (invariant-consistent-log-inverse state-Gks0 state-Gks0Map)
        (invariant-2e state-Gks0 state-Gks0Map)
        (invariant-5 state-Gks0 state-Gks0Map)
    )
)
```

Listing 12: Parts of the invariant file `SET.smt2`

4.2.5 One-sided invariants and invariant bubbling

BDEFKK prove invariance of state relation (2a) via a local argument over the Key and Log packages. A local argument simply means that state relation (2a) can be proved to be preserved across all oracle calls regardless of the choice of the caller oracle. The main reason is that state relation (2a) describes properties of only the state of Key and Log packages separately in each game. In other words, the properties hold for the composition $M_{\text{small}} := \texttt{Key} \circ \texttt{Log}$. In Section 2.5, we called the state relations that concern only one of the left and right games by *one-sided* state relations. If they are proved to be invariant, we called them *one-sided* invariants. For all packages N, one can prove due to the state separation of N and M_{small} that one-sided invariants of M_{small} are also one-sided invariants of $M_{\text{big}} := N \circ M_{\text{small}}$. That is, one-sided invariants bubble up from small packages to any bigger packages enclosing them. We formally state this property as follows:

Theorem 4.1 (Invariant bubbling theorem)**.** *Let I be a one-sided state relation expressing some properties about the state of SSP package M_{small}. Let N be another SSP package. Note that I is naturally a one-sided state relation (upto renaming of variables) for the package $M_{big} := N \circ M_{small}$. If I is a one-side invariant for M_{small} then I is also a one-sided invariant for M_{big}.*

The proof of theorem simply follows from the state separation of the packages in SSP. The package N can only modify the state of the package M_{small} by calling its oracles. On the other hand, any oracle query to the package M_{small} preserves the state relation. As a result, one can conclude that state relations are preserved after any oracle query to the package M_{big}.

Going back to our example, we can apply the theorem to $M_{\text{small}} := \texttt{Key} \circ \texttt{Log}$ while N can be the composition of all other packages. It then suffices to prove invariance of state relation (2a) for $M_{\text{small}} := \texttt{Key} \circ \texttt{Log}$. Notice that this is a stronger result than proving the invariance of (2a) for the games `Gks0` and `Gks0Map` because one proves that the state relation is preserved for all possible oracle queries with all possible arguments instead of specific values chosen by the enclosing package N.

We prove the invariance of state relations (2a) and (2e) in a separate proof file for the composition `Key` $\circ$ `Log`. We have separated each of the sub invariants (2a-i) to (2a-x) in order to make the proof more manageable and point out the dependencies between them. The following listing presents the proof file. All the SMT files for this proof exists on the same repository [Raj25b].

```
proof KeyLogInvariants {
    ...
    instance game_KeyLogGks0 = KeyLogGks0 {
        ...
    }
    instance game_KeyLogGks0Map = KeyLogGks0Map {
        ...
    }
    gamehops {
        equivalence game_KeyLogGks0 game_KeyLogGks0Map {
            SET: {
                invariant: [
                    ./proofs/abstract-functions.smt2
                    ./proofs/key-log/key-log-invariants.smt2
                    ./proofs/key-log/SET.smt2
                ]
                lemmas {
                    assert-J-invariants: [no-abort, assume-J-invariants]
                    assert-updated-invariant-log-inverse: [no-abort, assume-updated-
    invariant-log-inverse]
                    assert-invariant-2a-iii: [no-abort, assume-invariant-2a-iii, assume-
    updated-invariant-log-inverse, assume-J-invariants]
                    assert-invariant-consistent-log-for-dh-and-psk: [no-abort, assume-
    invariant-consistent-log-for-dh-and-psk, assume-invariant-2a-v, lemma-rand-is-eq]
                    assert-invariant-consistent-log-inverse-level-zero-psk: [no-abort,
    assume-invariant-consistent-log-inverse-level-zero-psk, assume-invariant-consistent-
    log-for-dh-and-psk]
```

```
                    assert-invariant-2a-ix-and-2a-x: [no-abort, assume-invariant-2a-ix-
    and-2a-x]
                    assert-invariant-2e: [no-abort, assume-invariant-2e]
                    assert-invariant-log-inverse-name: [no-abort, assume-invariant-log-
    inverse-name]
                    assert-invariant-log-preserves-name: [no-abort, assume-invariant-log-
    preserves-name, assume-invariant-log-inverse-name]
                    assert-invariant-2a-viii: [no-abort, assume-invariant-2a-viii, lemma-
    SAMPLE-output-length, assert-invariant-2a-iii, assert-invariant-log-preserves-name,
    lemma-injectivity-of-len_alg]
                    assert-invariant-2a-vii: [no-abort, assume-invariant-2a-vii]
                    assert-invariant-2a-iv: [no-abort, assume-invariant-2a-iv]
                    assert-invariants-2a-i-and-2a-ii: [no-abort, assume-invariants-2a-i-
    and-2a-ii]
                    assert-invariants-2a-v-and-2a-vi: [no-abort, assume-invariants-2a-v-
    and-2a-vi]
                    lemma-SET-returns-same-handle-for-non-dh-and-psk: [no-abort, assume-
    invariants-2a-v-and-2a-vi]
                }
            }
        }
    }
}
```

Although we are proving invariance of state relations with SSBee's game equivalence syntax but neither of the state relations express a property between the states of the left and right games. We refer the reader to the repository for the definition of each of *assume* and *assert* lemmata. Briefly, *assume* lemmata express the state relation for the states of the games before an oracle call and *assert* lemmata express the state relations for the states of the games after an oracle call. The only lemmata that are left without proof are `lemma-rand-is-eq` and `lemma-injectivity-of-len_alg`. The randomness mapping lemma is a common workaround for the randomness mapping universal quantification issue discussed in Section 2.5. Lemma `lemma-injectivity-of-len_alg` states that the abstract function `len_alg` is injective. This can be considered part of the theory we need for our uninterpreted function `len_alg`. Lemma `assert-J-invariants` proves the invariance of Invariant-J, lemma `assert-invariant-2e` proves the invariance of Invariant-2e, and lemma `assert-updated-invariant-log-inverse` proves the invariance of proeprties related to the inverse tables.

We want to highlight other state relations that have the potential to be proved similarly via the invariant bubbling theorem in a smaller context. State relations (2b), (2d), (3), and (4) are all one-sided state relations. However, the smallest package M_{small} for which the relations can be proved to be invariant is not necessarily determined by the tables involved in the relations. For example, state relation (4) can not be proved for the composition $\mathsf{Key} \circ \mathsf{Log}$ as it depends on how the oracles XPD and XTR construct the handles. We describe how deductive program verification tools can be possibly used to prove the invariance of one-sided state relations. As a result, proper integration of SSBee with other verification tools remains only two-sided state relations (2c), (5), and (6) to be proved in SSBee. (State relation (5) is partially proved for all the oracles except XPD and XTR.)

As the final note, our research on this project brought to the light that SSBee does not check state relations hold for the initial game states. Recall that proving initial left and right game states belong to the state relation is a crucial assumption for the Fundamental Theorem of Code Equivalence (2.3). We, though, have not verified this proof obligation in SSBee and leave it as a future work.

4.3 Cheat sheet of verification techniques for SSBee users

In this section, we enumerate a list of tips and techniques we used or discovered in this project to verify proof obligations or debug verifications.

Order of verification Start with same-output (first) and equal-aborts (second) properties and then proceed to the invariants! The reason is that it is easier for these two proof obligations to pinpoint where to focus on.

For same-output property one shall consider all the **return** statements of the left oracle and checks whether the right oracle returns the same value at some point. Usually there is also a correspondence between the return points. Similar to our approach, if the output of the oracles can be predicted, one can verify separately whether the left and right oracles return the correct values. This can speed up the SMT solver too if one proves the same-output property assuming these separate lemmata for output values of the left and right oracles. See, for example, the verification of same-out property for the oracles XTR and XPD at the repository [Raj25b].

For equal-aborts, one should focus on the assertions, abort commands, and the *Maybe* type unwrapping operation. All these operations can abort an oracle. For each of them, if they fail on the left game, there should be a corresponding assertion, abort command or unwrapping operation that fails on the right game and vice versa. Unwrapping operations are a common source of mistakes as they are implicit and hidden from the user in the package code. However, they are automatically inserted by SSBee in the compiled SMT-LIB file.

Abort! If you are trying to verify a lemma that assumes the built-in lemma `no-abort`, you can eliminate if-branches by putting **abort** in them. In other words, to debug your verification and find the required invariants, you can use **abort** in if-branches that you do not want to focus on when performing a step-by-step proof. If you put lemma `no-abort` as a dependency, SSBee assumes the oracles do no abort and program branches that include a **abort** command are automatically discarded.

Ghost variables Consider using ghost state variables (introduced in Section 2.5) to simplify your state relations and lemmata as shown in the KEM-DEM example in the same section. Since SMT solvers are better off dealing with quantifier-free formulae, ghost variables can help to remove unnecessary quantification. Precisely, ghost variables can store intermediary values of the state variables such as public/secret key pair or randomness strings used by the encryption schemes. Therefore, if you are modeling your encryption scheme functions with abstract functions and want to

express a specific property about them (e.g. scheme correctness), you can remove quantification over the public/secret key pair or randomness string by expressing the property with their concrete values stored in the ghost variables. Ghost variables can also store program counters to help you identify from which **return** statement the oracle returns or at which assertion or abort command the oracle fails. The idea is to define an integer state variable and assign unique values at interesting points of your oracle. You can then assert the counter to be specific value to ensure the oracle has the intended behavior. That is possible because SSBee returns the game state including all package states when it aborts.

Use lemmata Following the previous tip, consider using lemmata and minimize your direct assertions in the SMT solver. Although SSBee gives flexibility to the user for expressing properties in a rich language like SMT-LIB, this flexibility can be abused and lead to unknown results. Identify the dependencies of your lemma or code equivalence proof obligation (same-output, equal-aborts, and invariant) and state required properties as lemmata. For example, consider stating properties involving a general universal quantifier with the concrete (and possibly ghost) state variables or inputs of the oracles. See `lemmma-kem-correctness` of the KEM-DEM example in 2.5. Lemmata also brings visibility to the dependencies of properties we prove with SSBee and makes it easier to identify properties that are easier to prove with other verification tools.

Another application of lemmata are in proof debugging. If the SMT solver can not verify a compelx lemma, one can simulate a proof by case analysis by stating each case as lemma and try to prove the cases separately.

Assume randomness equality Consider directly assuming the randomness sampling equalities as a lemma if you have multiple randomness mapping for your oracle and you feel your invariants are strong and complete. We discussed the randomness mapping issue we faced in KEM-DEM example and our workaround for the issue in Section 2.6.6.

Lookup loops As mentioned earlier, SSBee does not support iteration statements and loops. We have introduced two techniques in this thesis to model a specific iteration scenario using abstract functions and tables. Both of these techniques try to solve the problem of finding a table entry with specific conditions. The first approach replaces the table lookup with a variant of epsilon operator inspired by Hilbert's epsilon calculus [AZ24]. See Lemma 5.20 in Section 5 verified in SSBee using this technique. The second approach is to define a separate *inverse* table T^{-1} and preemptively keep track of entries you are going to lookup in table T in future when updating table T. See Section 4.1.3 for such an example in SSBee translation of TLS 1.3 key schedule security games.

Simplify oracle code Simplifying should be an obvious and intuitive tip; however, we pinpoint some concrete examples. In security modeling of TLS 1.3 key schedule

we initially replaced irrelevant oracle calls for the code equivalence with a an abstract function. One example was the HASH(t) oracle exposed by the package `Hash`. We replace an oracle call to HASH(t) in the code of oracle XPD of the package `Xpd` with an abstract function `hash1(t)`. We argued in Section 4.1.2 that the HASH oracle does not abort if the hash algorithm is checked to be supported upfront in the package `Key`. After verifying some properties, we noticed that since the package $\texttt{Hash}^0$ is stateless (its state is only written in the real package but not in the idealized package $\texttt{Hash}^1$) and HASH oracle does not abort (i.e. a total mathematical function), we do not need to complicate the code by replacing the abstract function `hash1(t)` back again with the oracle. In general, if you can verify your lemmata by replacing a stateless abort-free oracle with an unrestricted abstract function, you have proved a stronger result and do not need to instantiate the abstract function with the concrete oracle code.

Taking one step further, this approach is also useful in debugging proofs. You may want to replace an even stateful oracle oracle with an abstract function and state some properties in a lemma only to focus on another challenging part of the oracle. This approach is generalized in the modular verification in program verification literature when method calls are replace with their postconditions.

Simplify games This is also another trivially-looking tip but can be extremely difficult or impossible in some situations. The idea is to pull out all the irrelevant code from the oracles of your games G_0 and G_1 that you want to prove code equivalent. The *pulling out* process can be formally described as a reduction to the games G'_0 and G'_1. (i.e. the irrelevant code are extracted into a reduction package $\mathcal{R}$ such that $G_b \overset{code}{\equiv} \mathcal{R} \rightarrow G'_b$) Then equivalence of G'_0 and G'_1 can be proved in SSBee. We have given a concrete example for this technique in Lemma 5.20 of Section 5.

Proving invariance In the following, we briefly describe several related techniques for proving invariance. The most practical technique is to isolate state relations you want to prove their invariance. Namely, if your state relation I is of the form $I = I_1 \wedge I_2 \wedge I_3 \wedge \cdots \wedge I_n$ (as is usually the case), one can try to prove I_j independently from the others when assuming I holds in the old game states. One can even take one step further and determine which state relations are necessary to prove I_j and only assume those. We have given an example of this approach in Section 4.2.5. Isolating invariants can make the verification much faster as the SMT solver does not need to search in a larger space. Moreover, similar to lemmata, it brings visibility to the dependency of the invariants. Inspired from the Section 4.2.5, try identifying the one-sided invariants and suitable subgames to apply the Invariant bubbling theorem 4.1. Since one-sided invariants are exclusive properties of either the left or the right game, they can be proved independently from the other game. With class analogy of packages in SSP, we are given a class (package) with a set of exposed methods (oracles) sharing the private state of the class and we want to prove some properties are preserved after each method call. We argue in our future vision of SSBee why we believe identifying one-sided invariants are useful.

Invariance of state relations with tables It is very common for the state relations to express a property with a universal quantifier for all table indices. (All state relations of Lemma C.2 in Section 4.2.4 are of this form) When proving invariance of such state relations, if the oracle does not modify the tables involved in the relation, then the preservation of the state relation is trivially followed. However, when the oracle modifies one or more entries of the table, these new entries might falsify the state relation in the new tables. One can dig deep into proof of invariance with the SMT-LIB language and identify what can or can not be proved by the SMT solver and correspondingly add necessary lemmata, change the code of packages or modify the state relations. We have given one such example of step by step invariance proving in the repository [Raj25d] of the SSBee project for verification of Lemma 5.20. As a final tip, it is best to define as general and strong as possible invariants.

Translation tips When translating the pseudocode of your SSP packages, consider defining stateless packages for reusing recurring pieces of code and increasing readability. Use `Bits(*)` to define complex data structures (e.g. recursive data types, see the discussion before Section 4.1.2) Use abstract functions for custom operations operations on your data structures but be mindful when using them. They can make life harder for the SMT solver.

SSBee output We want to emphasize that previous techniques are useful when we get *unknown* results from the SMT solver. If the SMT solver returns a *sat*, it has found a concrete counterexample and SSBee returns the inputs passed to the adversaries as well as a pair of possible game states before the oracle query!

4.4 Future vision for SSBee

In this section, we compare the current status of SSBee with a desired future version of SSBee.

Error reporting As mentioned before, for each lemma, SSBee tries to prove the lemma with a proper SMT-LIB formula passed to the SMT solver. If the SMT solver returns *unknown*, which is the case most of the time if the solver does not return *unsat*, the user does not receive any help for the source of the issue. For instance, proving the same-output lemma requires that if the left oracle returns a value v from one of its **return** , statements, the right oracle also returns the same v from some **return** statement. It might be the case that only one of the **return** statements are problematic. However, this information is not currently communicated to the user. A similar situation happens for the equal-aborts property where one **abort** statement, say on the left, aborts while the right oracle does not abort. This granualar information can help to user to focus their time and energy on the actual source of the problem. Currently, the idea to debug a failing same-output lemma is to insert **abort** statements before **return** statements to eliminate those cases. For the equal-aborts, one can not use the same idea but they can use ghost variables for program counters.

However, it is indeed possible to generate verification conditions for each **return** statement of one of the games and check which fails and which proves. Similarly, verification conditions can be generated for **abort** and **assert** statements.

Verification condition generation (also known as predicate transformer semantics) is a fundamental technique in program veirifcation to argue about correctness of programs. In fact, modern program verification tools translate high level programming languages to an Intermediate Verification Langauge (IVL) which is then verified using SMT solvers by the low level verifiers such as Viper [MSS16], Boogie [BCD+05], Dafny [Lei10], etc. The important property of the translation is soundeness: a verified translated program should imply correctness of the original program. Currently, SSBee translates to code of oracles directly to SMT-LIB langauge. SSBee translates each oracle to an SMT-LIB function that receives the oracle inputs and a game state and returns an output (abort or value) and a new game state using nested SMT-LIB ternary operators `ite`. As a result, the nesting depth of the `ite` operations grows with the number of `if` conditions in the oracle code. Moreover, SSBee inlines the code of queried oracles. These design choices treat an oracle as a big monolithic function that can be applied to some inputs and game state and returns an output or abort. This approach makes it difficult to argue about individual **return** , **assert** , or **abort** statements in the code.

Translation to intermediate verification languages or even adopting ideas for verification condition generation can significantly help the user with more granular errors. Moreover, directly relying on the SMT-LIB ternary operator `ite` makes it difficult to translate loops. In program verification, users of verification tools are required to provide invariants for their loops. Loop body is verified separately to check they preserve the invariants. After verification of the loop body, the loop is roughly replaced with its invariant condition. These steps rely on verification condition generation and as a result adopting the ideas of verification condition generation can help with adding support for loops.

Modular verification We mentioned in the cheat sheet that oracles codes can be simplified during proof debugging by replacing oracles with abstract functions and assuming some properties about them out-of-the-box as a lemma. This idea is a variant of modular verification. In modular verification approach to verification of methods that call other methods, firstly, a precondition Q and postcondition P is defined for each method M. In the next step, for each method M, assuming Q, we try to prove after execution of the method body Q holds. However, method calls in the body of M are not inline but rather only the precondition of a called method M' is checked at the call site and then the postcondition of the method is assumed at that point. The idea is similar to verification of loops where the body of the loop is verified separately that preserves the invariants and the whole loop is roughly replaced with an statement that assume the invariant holds at that point. We consider this feature very useful if the user can switch between modular verification and inlining when verifying lemmata in SSBee.

Integrations with other tools We introduced the concept of one-sided invariants in Section 2.14 and discussed the Invariant bubbling theorem (Theorem 4.1) in Section 4.2.4. Briefly, one-sided invariants are state properties of an SSP package that are preserved after each oracle query to the package. Analogy of SSP packages and classes translates the problem of proving invariance of one-sided state relations to proving invariance of some properties of a class private fields after each method call. Observe that we are only dealing with one class (one program) unlike our usual scenario of code equivalence between two packages (two programs). In our future vision of SSBee, it is conceivable that SSBee is integrated with program verification tools such as Dafny [Lei10] and Viper [MSS16] that are built for exactly the same problem of proving properties of a single program with many helpful program correctness debugging feature. As a result, we envision a successful integration of SSBee with these tools where, for example, Danfy or Viper code as well as a formal specification of the code equivalence proof obligations are automatically generated by SSBee and allowing the user to continue with the other tool and reuse the positive verification result back again in SSBee. Since one-sided invariants are very common (as can be seen in this thesis) in code equivalence proofs, we consider integration with program verification tools (and collaboration with the tool developers) a promising research direction helping cryptographers generating verified proofs easier.

Moreover, it is very helpful to generate lemma statements in other verification tools or even interactive theorem provers when a property can not be verified in SSBee. For example, SSBee is not very well suited for complex randomness mapping scenarios. Concretely, for a successful verification in SSBee, one needs corresponding randomness sampling operations from the left and right games because SSBee simply derandomizes the oracles and map the randomness strings they consume. However, the same randomness mapping concept may appear in more complex situations where there is no one to one correspondence between the sampling points of the left and right oracles. For example, distribution of uniformly sampling a string of length 128 is the same as concatenation of two uniformly sampled strings of length 64. (i.e. two randomness sampling operations from one oracle corresponding to one sampling operation in the other oracle). At the same time, EasyCrypt is another powerful verifier that has necessary probabilistic program logic to reason about such distributions. Another example is statistical game hops such as $Gstat^{b,grp}$ defined in Figure 35 and advantage of which is bounded in Lemma 5.4.

Finally, it is valuable to have the SMT solver output a proof for *unsat* results and check them automatically with a theorem prover engine such as Rocq [Roc25].

Proof editor SSBee is currently a command line tool. In a previous work [Pun21], Puniamurthy developed a proof viewer for SSP proofs and visualized the Yao's garbling scheme proof in the tool. His proof viewer allowed sketching SSP compositions as well as viewing packages, reductions and code equivalence proofs, lemmata, and theorems all in an organized visual and interactive environment in the web browser. As a follow-up on their work, we propose a SSP proof editor or integrated development environment (IDE) to write and type-check code of packages as well as sketching

SSP compositions while automating the proof of code equivalence and reductions with SSBee engine. Ideally, the tool should allow generation of boilerplate code for the compositions directly from the graphical environment. Moreover, it is desired that template codes for packages, proof files, invariants, randomness mappings, and lemmata are generated in a few clicks of the user. This environment could be implemented as a language extension for Visual Studio Code or other common IDEs.

5 Salted Oracle Diffie-Hellman Assumption Analysis

Brzuska, Delignat-Lavaud, Egger, Fournet, Kohbrok, Kohlweiss (BDEFKK) [BDLE+21] reduced key schedule security of TLS 1.3 to collision resistance of hash functions used in the protocol (SHA256, SHA384, SHA512), (dual) pseudorandomness of extract and expand functions, pre-image resistance of expand functions, and Salted Oracle Diffie-Hellman (SODH) assumption. (Refer to Section 3 for TLS 1.3 extractor and expanders.) SODH is inspired by the Oracle Diffie-Hellman (ODH) assumption introduced by Abdalla, Bellare, and Rogaway [ABR01] and the PRF-ODH assumption introduced by Jager, Kohlar, Schäge, and Schwenk (JKSS) [JKSS11].

The ODH assumption states that it is hard to distinguish $H(g^{uv})$ from a uniformly random string of the same length for a hash function H given g^u, g^v and even access to an oracle $\mathcal{H}_u(X) := H(X^u)$ where the adversary is disallowed from submitting $X = g^v$. Compared to ODH, SODH allows the adversary to choose a salt s for the extractor $\mathsf{xtr}(\cdot, s)$ where $\mathsf{xtr}(\cdot, s)$ replaces $H(\cdot)$. The salt is necessary in TLS 1.3 Key Schedule Security because the hash function in the assumption is in fact an extractor and as illustrated in Section 3, an extractor is used to combine the Diffie-Hellman (DH) secret and a salt derived from the Pre-shared Key (PSK). Therefore, it is important (and easier for the proof) to consider the salt is ultimately chosen by the adversary.

BFGJ [BFGJ17] introduce several variants of PRF-ODH assumption but in all of them the adversary is given the first DH share g^u and can make none, single or multiple queries (in different variants) to oracle $\mathsf{ODH}_u(X, C) := \mathsf{PRF}(X^u, C)$ where $\mathsf{PRF}(K, C)$ is a pseudorandom function. In the next stage, adversary is required to generate a challenge string c' and challenger responds with the second DH share g^v and either $\mathsf{PRF}(g^{uv}, c')$ or a uniformly random string of the same length. After this point and before returning its guess, the adversary is allowed to make none, single or multiple queries (in different variants) to oracle $\mathsf{ODH}_v(X, C) := \mathsf{PRF}(X^v, C)$. One can think of c' as salt chosen by the adversary.

Unlike the PRF-ODH, the adversary in the SODH game is allowed to receive as many honest DH shares as it wishes (DH share $X = g^x$ is honest when x is uniformly chosen by the challenger and is not known to the adversary) and observes $\mathsf{xtr}(Y^x, s)$ upon querying any adversarially chosen salt s, any adversarially chosen group element Y, and honest DH share $X = g^x$ generated by the security game. Moreover, the adversary in the SODH can freely interleave these oracle calls from the beginning without being restricted to choose the salt based on one DH share. Finally, the adversary is tasked with distinguishing $\mathsf{xtr}(g^{xy}, s)$ from a uniformly random string of the same length where x and y are private exponents of honest DH shares $X = g^x$ and $Y = g^y$.

Another difference between SODH and PRF-ODH is that SODH allows the hash function to be agile. Recall from Section 3, $\mathsf{xtr}(\cdot, s)$ is based on HMAC which in turn is based on the hash function. The TLS 1.3 standard allows the use of three hash algorithms SHA256, SHA384, and SHA512. Due to the possibility of the clients and servers in TLS sessions to hash the same DH secret under different algorithms, the

SODH assumption too, allows the adversary to choose the hash algorithm under which $\mathsf{xtr}(Y^x, s)$ is computed by "tagging" the salt (i.e. using a salt length corresponding to the hash algorithm).

BDEFKK [BDLE+21] optimistically claim that SODH assumption can be reduced to computational Diffie-Hellman (CDH) assumption in the random oracle model. In this section, we show that SODH assumption can be reduced in the programmable random oracle model to the well-known strong Diffie-Hellman (SDH) assumption also introduced by Abdalla, Bellare, and Rogaway [ABR01]. SDH assumption is the same as CDH assumption (computing $g^{\alpha\beta}$ given g^{α} and g^{β}) except that the adversary has additionally access to a variant of decisional Diffie-Hellman (DDH) oracle $\mathsf{DDH}_{\alpha}(X, Y) := (X^{\alpha} \stackrel{?}{=} Y)$ where g^{α} is one of the DH shares returned by CDH challenger. The additional DDH oracle is helpful in checking random oracle outputs consistency with other oracles in SODH security game.

Additionally, our security analysis also shows that SODH assumption captures another security notion with roots in the natural collisions of DH secrets described in Section 3.1.1. To describe this collisions, let DH shares $X = g^x$, $X' = g^{x'}$, and $Y = g^y$ are honest shares chosen by the SODH game where the exponents x, x', and y are unknown the adversary. If the adversary can find a share $Y' = g^{y'}$ such that $Y'^{x'} = Y^x$ (i.e. DH secrets of pairs (X, Y) and (X', Y') collide) then the adversary can break SODH game without necessarily computing Y^x. That is due to the fact that $\mathsf{xtr}(Y^x, s) = \mathsf{xtr}(Y'^{x'}, s)$ but upon adversary's query, the ideal SODH game samples and returns a random string instead of $\mathsf{xtr}(Y^x, s)$ due to the honesty of Y^x. (X and Y are both chosen by the adversary) At the same time, $\mathsf{xtr}(Y'^{x'}, s)$ is directly returned to the adversary upon its query in the ideal game. Therefore, a reduction from SODH to CDH shall also consider these cross collision attacks. We will later on formalize this security notion and reduce SODH to a combination of SDH and this notion.

Remark. Interestingly, BFGJ [BFGJ17] show an impossibility result that existence of a black-box algebraic reduction [20] in the standard model from snPRF-ODH or nsPRF-ODH [21] to a DDH-augmented problem [22] implies either the DDH-augmented problem is not hard or decisional square Diffie-Hellman problem is not hard. Although snPRF-ODH is a weaker assumption than SODH, the impossibility shows the importance of the random oracle. However, in the random oracle model making the random oracle outputs to be consistent with responses of other oracles is very difficult when reducing from SODH to CDH because the reduction does not have access to a DDH oracle anymore.

Inspired by mmPRF-ODH instantiation of [BFGJ17], we reduce SODH to SDH in two steps. First, we reduce the SODH assumption to the augmented square Diffie-Hellman assumption (SqDH). The augmented SqDH assumption is the same as the square Diffie-Hellman assumption, where the adversary's task is to compute

[20] any reduction that performs group operations in a pre-defined way such as multiplication

[21] snPRF-ODH is a variant of PRF-ODH in which the adversary can make only a single query to $\mathsf{ODH}_u(X, C)$ but no queries to $\mathsf{ODH}_v(X, C)$

[22] a combination of an abstract cryptographic problem and a DDH problem such that a solution to either of them is a solution to the augmented problem

g^{α^2} given g^{α}, except that the adversary is additionally given access to a DDH oracle $\mathsf{DDH}_{\alpha}(X, Y) := (X^{\alpha} \stackrel{?}{=} Y)$. In the next step, SqDH is reduced to SDH, via the original reduction by [BFGJ17]. We refer the reader to Section 3.3 of their work for proof of this step.

Section 5.1 formally defines the security games for SqDH, SDH, SODH, and RO-SODH (random oracle variant of SODH) assumptions using the State-separating proofs (SSP) framework. Section 5.2 presents the security reduction.

5.1 Security games

We define SODH, SODH-RO, SqDH, SDH games as follows:

$$
\begin{aligned}
Gsodh^{b,grp} &:= \mathtt{SodhCore}^{b,grp} \longrightarrow \mathtt{xtr0} \\
Gsodhro^{b,grp} &:= \frac{\mathtt{SodhCore}^{b,grp}}{\mathtt{ID}_H} \longrightarrow \mathtt{RO} \\
Gsqdh^{b,grp} &:= \mathtt{Sqdh}^{b,grp} \\
Gsdh^{b,grp} &:= \mathtt{Sdh}^{b,grp}
\end{aligned}
$$

where packages `SodhCore`, `RO`, `xtr0`, `Sqdh`, `Sdh` are defined in Figure 34. Figure 33 visualizes the call graph of games $Gsodh^{b,grp}$ and $Gsodhro^{b,grp}$.

(a) $Gsodh^{b,grp}$ **(b)** $Gsodhro^{b,grp}$

Figure 33: SODH and SODH-RO games

Similar to the TLS key schedule security, we consider concrete security assumptions as described in Section 2.1. Briefly, we relate the advantage of an adversary playing in $Gsodh^{b,grp}$ to the advantage of adversary in $Gsdh^{b,grp}$ game.

The SODH game $Gsodh^{b,grp}$ after inlining code of `xtr0` package matches exactly the game defined in Section 3.3 of [BDLE+21]. Notice that a DH share X is honest when it is returned by $\mathsf{DHGEN}()$ and $E[X]$ is set. Observe that $\mathsf{XTR}(X, Y, s)$ computes $\mathsf{xtr}(Y^x, s)$ when $X = g^x$ is an honest share generated by $\mathsf{DHGEN}()$ by asserting $E[X] \neq \perp$ in the very first line. Moreover, the only difference between the real and ideal SODH game is when the adversary queries $\mathsf{XTR}(X, Y, s)$ with X and Y both being honest. The ideal game uniformly samples a bitstring of length $\mathsf{len}(salt)$. Notice that it is crucial for Y to be honest when a random string is sampled in the ideal game.

When $\mathsf{xtr}(\cdot, \cdot)$ is modeled as a random oracle, all outputs by $\mathsf{XTR}(X, Y, s)$ and random oracle $\mathsf{H}(Z, s)$ are uniformly random strings in game SODH-RO. Hence, the only way for the adversary to distinguish the real and ideal game is to query

$\underline{\texttt{SodhCore}^{b,grp}}$

Parameters

b : idealization bit
grp : group description

State

E : table
S : table

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
$x \leftarrow\!\!\$\ Z_q$
$X \leftarrow g^x$
$E[X] \leftarrow x$
return X

XTR$(X, Y, salt)$

assert $E[X] \neq \perp \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
$alg \leftarrow \mathsf{alg}(salt)$
if $b \wedge E[Y] \neq \perp$:
  $h \leftarrow \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$
  if $S[h, salt] = \perp$:
    $S[h, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
  return $S[h, salt]$
return $\mathsf{H}(Y^{E[X]}, salt)$

$\underline{\texttt{RO}}$

State

T : table

H$(Z, salt)$

if $T[Z, salt] \neq \perp$:
  return $T[Z, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T[Z, salt]$

$\underline{\texttt{Sqdh}^{b,grp}}$

Parameters

b : idealization bit
grp : group description

State

α : integer

SAMPLE()

$g \leftarrow \mathsf{gen}(grp)$
if $\alpha \neq \perp$:
  return g^α
$q \leftarrow \mathsf{ord}(g)$
$\alpha \leftarrow\!\!\$\ Z_q$
return g^α

CHECK(Z)

assert $\alpha \neq \perp$
$g \leftarrow \mathsf{gen}(grp)$
if $b \wedge Z = g^{\alpha^2}$:
  return true
return false

DDH(Y, Z)

assert $\alpha \neq \perp$
if $Y^\alpha = Z$:
  return true
return false

$\underline{\texttt{xtr0}}$

H$(Z, salt)$

$alg \leftarrow \mathsf{alg}(salt)$
return $\mathsf{xtr}_{alg}(Z, salt)$

$\underline{\texttt{Sdh}^{b,grp}}$

Parameters

b : idealization bit
grp : group description

State

α : integer
β : integer

SAMPLE()

$g \leftarrow \mathsf{gen}(grp)$
if $\alpha \neq \perp$:
  return g^α
$q \leftarrow \mathsf{ord}(g)$
$\alpha \leftarrow\!\!\$\ Z_q$
$\beta \leftarrow\!\!\$\ Z_q$
return (g^α, g^β)

CHECK(Z)

assert $\alpha \neq \perp$
$g \leftarrow \mathsf{gen}(grp)$
if $b \wedge Z = g^{\alpha\beta}$:
  return true
return false

DDH(Y, Z)

assert $\alpha \neq \perp$
if $Y^\alpha = Z$:
  return true
return false

Figure 34: Code of packages SodhCore, xtr0, RO, Sqdh, Sdh

$\mathsf{XTR}(X, Y, s)$ with $X = g^x$ and $Y = g^y$ both being honest and also the random oracle on $\mathsf{H}(g^{xy}, s)$. In the real game, the answers are consistent while in the ideal game they are not except with probability $\frac{1}{2^{\mathsf{len}(s)}}$. In the ideal game, the oracle $\mathsf{XTR}(X, Y, s)$ samples a random string r and store it in the table S while the oracle $\mathsf{H}(g^{xy}, s)$ only searches the table T and sample another random string independently from r.

We encode search-based SDH game from Section 5 of [ABR01] as an indistinguishability game in SSP using CHECK oracle. (See 2.1 for a discussion on search games.) SqDH search-based game from Section 3.3 of [BFGJ17] is also encoded as an indistinguishability game.

Notice that $salt$ is agile and tagged with the description of a hash function algorithm from the set $\mathcal{H} = \{\mathtt{sha256}, \mathtt{sha384}, \mathtt{sha512}\}$. Therefore, $\mathsf{len}(alg) \in \{256, 384, 512\}$ for $alg \in \mathcal{H}$. Moreover, the extractor algorithm $\mathsf{xtr}_{alg}(\cdot, \cdot)$ is also agile. It checks the tag of the salt and use the corresponding hash function for the underlying hmac operation. Also, $\mathsf{gen}(grp)$ returns the generator of group g, $\mathsf{ord}(g)$ returns the order of generator, and $\mathsf{id}(grp)$ returns the identity element of the group. [23] Z_q denotes the positive integers less than or equal to q. Let $\langle X \rangle$ be a bitstring representation of a group element X. Handle $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$ can be defined to be the tuple (X, Y) if $\langle X \rangle < \langle Y \rangle$ and (Y, X) otherwise. Finally, $T[K]$ denotes entry in the table T with key K.

5.2 Security reduction

As hinted in Section 3.2, $\mathsf{sort}(X, Y)$ is critical in the game $Gsodhro^{b,grp}$ to prevent trivial attacks by swapping the shares. Namely, without sorting the shares, two distinct handles $\mathsf{dh}\langle X, Y\rangle$ and $\mathsf{dh}\langle Y, X\rangle$ point to the same key g^{xy} that confuses the ideal game to sample distinct strings of length $\mathsf{len}(alg)$ with high probability. This can be easily exploited by the adversary to distinguish the real and ideal games. However, sorting only prevents trivial collision attacks. We refer to a pair of DH shares (X, Y) as honest if they are both generated by the oracle DHGEN. We denote the honesty of the pair by $hon(X, Y)$ which can be 0 (dishonest) or 1 (honest). We say two pairs of DH shares $(X = g^x, Y = g^y)$ and $(X' = g^{x'}, Y' = g^{y'})$ collide if $g^{xy} = g^{x'y'}$. If the adversary can cause a collision between two pairs (X, Y) and (X', Y') where $hon(X, Y) = hon(X', Y') = 1$ (two honest pairs) or $hon(X, Y) \neq hon(X', Y')$ (an honest and a dishonest pair), it can distinguish the real and ideal games $Gsodhro^{0,grp}$ and $Gsodhro^{1,grp}$. In case of $hon(X, Y) = hon(X', Y') = 1$, the ideal game samples two different strings for essentially the same value $\mathsf{xtr}(g^{xy}, s)$ for adversarially chosen s. We define the pair of security games $Ghcoll^{b,grp}$ for $b \in \{0, 1\}$ and group grp in Figure 35 to capture these honest collisions. In case of $hon(X, Y) \neq hon(X', Y')$, say Y' is dishonest, the adversary distinguish the real and ideal games by choosing some salt s and querying $\mathsf{XTR}(X, Y, s)$ and $\mathsf{XTR}(X', Y', s)$. Query $\mathsf{XTR}(X, Y, s)$ is responded with a uniformly random string while $\mathsf{XTR}(X', Y', s)$ is responded with

[23] Order of generator is the same as order of group if the group is cyclic of prime order. The order of group being prime is very important for the statistical game hop as well as last game hop we make in the proof of Lemma 5.6. See [CJ19] for small subgroup and invalid curve attacks on protocols using nonprime-ordered Diffie-Hellman groups.

$\mathsf{xtr}(Y'^{x}, s)$ which collides with the randomly chosen string with small probability. We define the pair of security games $Gccoll^{b,grp}$ for $b \in \{0,1\}$ and group grp in Figure 36 to capture these cross collisions. As hinted before, we first reduce SODH assumption to SqDH and then reduce SqDH to SDH. Notice that these collision attacks prevent an adversary $\mathcal{A}$ against $Gsodhro^{b,grp}$ to be used to break $Gsqdh^{b,grp}$ because the adversary $\mathcal{A}$ can distinguish the games $Gsodhro^{0,grp}$ and $Gsodhro^{1,grp}$ without querying the random oracle on the SqDH secret g^{α^2}. We bound the advantage of any adversary $\mathcal{A}$ against the game in Lemma 5.1. On the other hand, game $Gccoll^{b,grp}$ is the other security notion captured by SODH that our analysis brought to the light. In Theorem 5.2, we reduce SODH to indistinguishability of the games $Gccoll^{b,grp}$ and $Gsdh^{b,grp}$.

Lemma 5.1. *For any adversary $\mathcal{A}$, $\mathsf{Adv}(\mathcal{A}, Ghcoll^{b,grp}) \leq \frac{6\binom{Q}{2}}{q} + \frac{36\binom{Q}{3}}{q^2} + \frac{24\binom{Q}{4}}{q^3}$.*

Proof. See Section 5.2.1. □

Theorem 5.2 (SDH implies SODH)**.** *Let $\mathcal{A}$ be an adversary. Then, there exists PPT reductions $\mathcal{R}^{grp}$, $\mathcal{R}_{sdh}$, and $\mathcal{R}^{b,grp}_{ccoll}$ such that,*

$$\begin{aligned}\mathsf{Adv}(\mathcal{A}, Gsodhro^{b,grp}) \leq \sqrt{\mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{grp} \to \mathcal{R}_{sdh}, Gsdh^{b,grp})} \\ + 2 \times \left(\frac{6\binom{Q}{2}}{q} + \frac{36\binom{Q}{3}}{q^2} + \frac{24\binom{Q}{4}}{q^3} + \frac{1}{q}\right) \\ + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{0,grp}_{ccoll}, Gccoll^{b,grp}) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{1,grp}_{ccoll}, Gccoll^{b,grp})\end{aligned}$$

where Q is the number of DHGEN *queries made by the adversary and q is the order of group grp.*

In order to prove Theorem 5.2, we define two auxiliary games $Gstat^{b,grp}$ and $Galpha^{b,grp}$. Figures 37 and 38 respectively presents the pseudocode definition of the games $Gstat^{b,grp}$ and $Galpha^{b,grp}$. We bound the advantage of any adversary $\mathcal{A}$ against the games $Galpha^{b,grp}$ and $Gstat^{b,grp}$ respectively in Lemma 5.3 and Lemma 5.4.

Lemma 5.3. *For any adversary $\mathcal{A}$, $\mathsf{Adv}(\mathcal{A}, Galpha^{b,grp}) \leq \frac{1}{q}$ where q is the order of group grp.*

Proof. See Section 5.2.2. □

Lemma 5.4. *For any adversary $\mathcal{A}$, $\mathsf{Adv}(\mathcal{A}, Gstat^{b,grp}) = 0$.*

Proof. See Section 5.2.3. □

Next, we adopt the following result from [BFGJ17].

Lemma 5.5 (SDH implies SqDH, proved in Section 3.3 of [BFGJ17])**.** *Let $\mathcal{A}$ be an adversary. Then, there exists a PPT reduction $\mathcal{R}_{sdh}$ such that,*

$$\mathsf{Adv}(\mathcal{A}, Gsqdh^{b,grp}) \leq \sqrt{\mathsf{Adv}(\mathcal{A} \to \mathcal{R}_{sdh}, Gsdh^{b,grp})}$$

$\mathcal{R}^{b,grp}_{hcoll}$

State

S : table

T : table

DHGEN()

return DHGEN()

XTR($X, Y, salt$)

assert DHGET(X) $\neq \perp \wedge$
  grp(X) = grp(Y) = $grp \wedge$
  alg($salt$) $\in \mathcal{H}$
$alg \leftarrow$ alg($salt$)
if $b \wedge$ DHGET(Y) $\neq \perp$:
  $f \leftarrow$ FIND($X, Y, salt$) :
  if $f \neq \perp$:
    return f
  if $S[X, Y, salt] \neq \perp$:
    return $S[X, Y, salt]$
  if $S[Y, X, salt] \neq \perp$:
    return $S[Y, X, salt]$
  $S[X, Y, salt] \leftarrow\$ \{0,1\}^{\text{len}(alg)}$
  SET($X, Y, salt, S[X, Y, salt]$)
  return $S[X, Y, salt]$
return H($Y^{\text{DHGET}(X)}, salt$)

H($Z, salt$)

if $T[Z, salt] \neq \perp$:
  return $T[Z, salt]$
$alg \leftarrow$ alg($salt$)
$T[Z, salt] \leftarrow\$ \{0,1\}^{\text{len}(alg)}$
return $T[Z, salt]$

$G^{b,grp}_{hcoll}$

Parameters

b : idealization bit

grp : group description

State

S : table

DHGEN()

$g \leftarrow$ gen(grp)
$q \leftarrow$ ord(g)
$x \leftarrow\$ Z_q$
$X \leftarrow g^x$
$E[X] \leftarrow x$
return X

DHGET(X)

return $E[X]$

SET($X, Y, salt, h$)

assert $E[X] \neq \perp \wedge E[Y] \neq \perp$
$S[X, Y, salt] \leftarrow h$

FIND($X, Y, salt$)

assert $E[X] \neq \perp \wedge E[Y] \neq \perp$
if b :
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]} \wedge$
        $\{X, Y\} \neq \{X', Y'\}$:
      return $S[X', Y', salt]$
return $\perp$

$G^{b,grp}_{3}$

State

E : table

S : table

T : table

DHGEN()

same as the left

XTR($X, Y, salt$)

assert $E[X] \neq \perp \wedge$
  grp(X) = grp(Y) = $grp \wedge$
  alg($salt$) $\in \mathcal{H}$
$alg \leftarrow$ alg($salt$)
if $b \wedge E[Y] \neq \perp$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]} \wedge$
        $\{X, Y\} \neq \{X', Y'\}$:
      return $S[X', Y', salt]$
  if $S[X, Y, salt] \neq \perp$:
    return $S[X, Y, salt]$
  if $S[Y, X, salt] \neq \perp$:
    return $S[Y, X, salt]$
  $S[X, Y, salt] \leftarrow\$ \{0,1\}^{\text{len}(alg)}$
  return $S[X, Y, salt]$
return H($Y^{E[X]}, salt$)

H($Z, salt$)

same as the left

Figure 35: Game $Ghcoll^{b,grp}$, reduction $\mathcal{R}^{b,grp}_{hcoll}$, and game $G^{b,grp}_{3}$ for comparison

$\mathcal{R}^{b,grp}_{ccoll}$

DHGEN()

$X \leftarrow \mathsf{DHGEN}()$
$E[X] \leftarrow X$
return X

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
$alg \leftarrow \mathsf{alg}(salt)$
$f \leftarrow \mathsf{FIND}(X, Y, salt)$
if $f \neq \bot$:
  return f
if $b \wedge E[Y] \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $\mathsf{CLAW}(Y, X, Y', X')$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
  $\mathsf{SET}(X, Y, salt, S[X, Y, salt])$
  return $S[X, Y, salt]$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $\mathsf{CLAW}(Y, X, Y', X')$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
foreach $(Z, salt)$ **in** T_{H} :
  if $\mathsf{DDH}(X, Y, Z)$:
    return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
$\mathsf{SET}(X, Y, salt, T_{\mathsf{XTR}}[X, Y, salt])$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $\mathsf{DDH}(X, Y, Z)$:
    return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

$Gccoll^{b}$

Parameters

b : idealization bit
grp : group description

State

L : table
E : table

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
$x \leftarrow\!\!\$\ Z_q$
$X \leftarrow g^x$
$E[X] \leftarrow x$
return X

SET$(X, Y, salt, h)$

assert $E[X] \neq \bot$
$L[X, Y, salt] \leftarrow h$

FIND$(X, Y, salt)$

assert $E[X] \neq \bot$
if $\neg b$:
  return $\bot$
foreach $(X', Y', salt)$ **in** L :
  if $Y^{E[X]} = Y'^{E[X']} \wedge$
  $\big((E[Y] = \bot) \neq (E[Y'] = \bot)\big)$:
    return $L[X', Y', salt]$
return $\bot$

DDH(X, Y, Z)

assert $E[X] \neq \bot$
return $Y^{E[X]} = Z$

CLAW(Y, X, Y', X')

assert $E[X] \neq \bot \wedge E[X'] \neq \bot$
return $Y^{E[X]} = Y'^{E[X']}$

$G^{b,grp}_{7}$

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** L :
  if $Y'^{E[X']} = Y^{E[X]} \wedge$
  $\big((E[Y] = \bot) \neq (E[Y'] = \bot)\big)$:
    return $L[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
if $b \wedge E[Y] \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]}$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
  $L[X, Y, salt] \leftarrow S[X, Y, salt]$
  return $L[X, Y, salt]$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
foreach $(Z, salt)$ **in** T_{H} :
  if $Y^{E[X]} = Z$:
    return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
$L[X, Y, salt] \leftarrow T_{\mathsf{XTR}}[X, Y, salt]$
return $L[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $Y^{E[X]} = Z$:
    return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

Figure 36: Game $Gccoll^{b,grp}$, reduction $\mathcal{R}^{b,grp}_{ccoll}$, and game $G^{b,grp}_{7}$ for comparison

$\mathcal{R}_{stat}^{b,grp}$

State

T_{H} : table

T_{XTR} : table

DHGEN()

return DHGEN()

XTR$(X, Y, salt)$

assert DHGET$(X) \neq \perp \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{\mathsf{DHGET}(X')} = Y^{\mathsf{DHGET}(X)}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(Z, salt)$ **in** T_{H} :
  if $Y^{\mathsf{DHGET}(X)} = Z$:
    if DHGET$(Y) = \perp \vee \neg b$:
      return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \perp$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $Y^{\mathsf{DHGET}(X)} = Z$:
    if DHGET$(Y) = \perp \vee \neg b$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

$Gstat^{b,grp}$

Parameters

b : idealization bit

grp : group description

State

E : table

α : integer

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
$x \leftarrow\!\!\$\ Z_q$
if b :
  if $\alpha = \perp$:
    $\alpha \leftarrow\!\!\$\ Z_q$
  $e \leftarrow \alpha x$
else :
  $e \leftarrow x$
$X \leftarrow g^e$
$E[X] \leftarrow e$
return X

DHGET(X)

return $E[X]$

$G_9^{b,grp}$

State

E : table

T_{H} : table

T_{XTR} : table

α : integer

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
if $\alpha = \perp$:
  $\alpha \leftarrow\!\!\$\ Z_q$
$x \leftarrow\!\!\$\ Z_q$
$X \leftarrow g^{\alpha x}$
$E[X] \leftarrow \alpha x$
return X

XTR$(X, Y, salt)$

same as the left

H$(Z, salt)$

same as the left

Figure 37: Game $Gstat^{b,grp}$, reduction $\mathcal{R}_{stat}^{b,grp}$, and game $G_9^{b,grp}$ for comparison

$\mathcal{R}^{b,grp}_{alpha}$

State

E : table

T_{H} : table

T_{XTR} : table

α : integer

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$

$q \leftarrow \mathsf{ord}(g)$

$\alpha \leftarrow \mathsf{SAMPLE}()$

$x \leftarrow\!\!\$\ Z_q$

$X \leftarrow (g^x)^\alpha$

$E[X] \leftarrow x$

return X

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$

foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $\mathsf{CHECK}(Y'^{E[X']}, Y^{E[X]})$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$

$alg \leftarrow \mathsf{alg}(salt)$

foreach $(Z, salt)$ **in** T_{H} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{H}}[Z, salt]$

$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$

foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$

$alg \leftarrow \mathsf{alg}(salt)$

$T_{\mathsf{H}}[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $T_{\mathsf{H}}[Z, salt]$

$Galpha^{b,grp}$

Parameters

b : idealization bit

grp : group description

State

α : integer

SAMPLE()

$g \leftarrow \mathsf{gen}(grp)$

$q \leftarrow \mathsf{ord}(g)$

if $\alpha = \bot$:
  $\alpha \leftarrow\!\!\$\ Z_q$

if b :
  assert $\alpha \neq q$

return α

CHECK(S, T)

if b :
  return $S = T$

return $S^\alpha = T^\alpha$

$G^{b,grp}_{11}$

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$

$q \leftarrow \mathsf{ord}(g)$

if $\alpha = \bot$:
  $\alpha \leftarrow\!\!\$\ Z_q$

assert $\alpha \neq q$

$x \leftarrow\!\!\$\ Z_q$

$X \leftarrow (g^\alpha)^x$

$E[X] \leftarrow x$

return X

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$

foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$

$alg \leftarrow \mathsf{alg}(salt)$

foreach $(Z, salt)$ **in** T_{H} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{H}}[Z, salt]$

$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$

foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$

$alg \leftarrow \mathsf{alg}(salt)$

$T_{\mathsf{H}}[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $T_{\mathsf{H}}[Z, salt]$

Figure 38: Game $Galpha^{b,grp}$, reduction $\mathcal{R}^{b,grp}_{alpha}$, and game $G^{b,grp}_{11}$ for comparison

As the final step towards proving Theorem 5.2, we reduce SODH to SqDH.

Lemma 5.6 (SqDH implies SODH)**.** *Let $\mathcal{A}$ be an adversary. Then, there exists PPT reductions $\mathcal{R}^{grp}$, $\mathcal{R}^{b,grp}_{ccoll}$, $\mathcal{R}^{b,grp}_{stat}$, $\mathcal{R}^{b,grp}_{hcoll}$, and $\mathcal{R}^{b,grp}_{alpha}$ such that,*

$$\begin{aligned}\mathsf{Adv}(\mathcal{A}, Gsodhro^{b,grp}) \leq\ & \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{grp}, Gsqdh^{b,grp}) \\ & + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{0,grp}_{ccoll}, Gccoll^{b,grp}) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{1,grp}_{ccoll}, Gccoll^{b,grp}) \\ & + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{0,grp}_{stat}, Gstat^{b,grp}) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{1,grp}_{stat}, Gstat^{b,grp}) \\ & + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{0,grp}_{hcoll}, Ghcoll^{b,grp}) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{1,grp}_{hcoll}, Ghcoll^{b,grp}) \\ & + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{0,grp}_{alpha}, Galpha^{b,grp}) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{1,grp}_{alpha}, Galpha^{b,grp})\end{aligned}$$

Proof of Theorem 5.2. Theorem follows from application of Lemmata 5.6, 5.1, 5.4, and 5.3. □

We define the pseudocode of our main reduction $\mathcal{R}^{grp}$ in Figure 39 and use reduction $\mathcal{R}^{grp}$ to present the high level idea of the proof of Lemma 5.6.

Proof idea of Lemma 5.6. Having an adversary $\mathcal{A}$ breaking the security of SODH assumption, we want to build an adversary $\mathcal{B}$ to break the security of SqDH assumption. To this end, we want to simulate SODH game using a reduction $\mathcal{R}^{grp}$ interacting with SqDH game. As a result of a correct simulating reduction $\mathcal{R}^{grp}$, we build the new adversary $\mathcal{B} := \mathcal{A} \to \mathcal{R}^{grp}$ as a composition of $\mathcal{A}$ and $\mathcal{R}^{grp}$. Reduction works as follows. Receiving a single DH share g^{α} from $\mathsf{SAMPLE}()$ oracle of SqDH game, we need to generate as many shares as the adversary requires in SODH game when calling $\mathsf{DHGEN}()$. We achieve this by sampling a DH share (say g^{α} for random $\alpha \leftarrow\!\!\$\ Z_q$) by calling $\mathsf{SAMPLE}()$ and for each $\mathsf{DHGEN}()$ call, we uniformly sample a fresh random $x \leftarrow\!\!\$\ Z_q$, store $E[X] \leftarrow x$ for $X = (g^{\alpha})^x$ (instead of $E[X] \leftarrow \alpha x$) and return $(g^{\alpha})^x$ to the adversary. Using the group being prime-ordered, we notice that distribution of $g^{\alpha x_i}$ for uniform α and x_i's is the same as g^{x_i} for uniform x_i's. (See Lemma 5.4.) However, having stored x instead of αx in the exponent table, we cannot compute $\mathsf{xtr}(Y^{E[X]}, salt)$ when the adversary calls $\mathsf{XTR}(X, Y, salt)$. Instead, taking advantage of $\mathsf{xtr}(\cdot,\cdot)$ being modeled as a random oracle $\mathsf{H}(Z, salt)$, we sample a random bitstring $s \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$ and program the random oracle at $\mathsf{xtr}(Y^{E[X]}, salt)$ and set the programmed response at $T_{\mathsf{XTR}}[X, Y, salt] \leftarrow s$. In turn, we need to respond consistently when the random oracle is queried on $\mathsf{H}(Z, salt)$ where $Z = Y^{E[X]}$ but $T_{\mathsf{XTR}}[X, Y, salt] \neq \perp$ for some group elements X and Y. Interestingly, the consistency can be checked using the $\mathsf{DDH}(Y, Z)$ oracle provided by the SqDH game. Furthermore, we need to make $\mathsf{XTR}(X, Y, salt)$ queries consistent as it is possible for the adversary to query $\mathsf{XTR}(X, Y, salt)$ and $\mathsf{XTR}(X', Y', salt)$ where $Y'^{E[X']} = Y^{E[X]}$. These checks, however, can be performed when knowing only x and x' while $X = g^{\alpha x}$ and $X' = g^{\alpha x'}$ since $Y'^{x'} = Y^{x}$ is implied by $Y'^{\alpha x'} = Y^{\alpha x}$ in a prime-ordered group except when $\alpha = q$, which is chosen with probability $\frac{1}{q}$. (See Lemma 5.3.) Notice that if all shares X, Y, X', Y' are honest but $\{X, Y\} \neq \{X', Y'\}$ (i.e. $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle \neq \mathsf{dh}\langle\mathsf{sort}(Y, X)\rangle$), $\mathsf{XTR}(X, Y, salt)$ and $\mathsf{XTR}(X', Y', salt)$ both sample a new string and they are only

$\mathcal{R}^{grp}$

Parameters

grp : group description

State

E : table
T_{H} : table
T_{XTR} : table
A : group element

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
$A \leftarrow\!\!\$\ \mathsf{SAMPLE}()$
assert $A \neq g^q$
$x \leftarrow\!\!\$\ Z_q$
$X \leftarrow A^x$
$E[X] \leftarrow x$
return X

XTR($X, Y, salt$)

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(Z, salt)$ **in** T_{H} :
  if $\mathsf{DDH}(Y^{E[X]}, Z)$:
    if $E[Y] = \bot \vee \neg\mathsf{CHECK}(Z^{E[X]^{-1}E[Y]^{-1}})$:
      return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H($Z, salt$)

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $\mathsf{DDH}(Y^{E[X]}, Z)$:
    if $E[Y] = \bot \vee \neg\mathsf{CHECK}(Z^{E[X]^{-1}E[Y]^{-1}})$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

Figure 39: Code of reduction $\mathcal{R}^{grp}$

the same with probability $\frac{1}{2^{\mathsf{len}(alg)}}$. Therefore, the outputs are inconsistent and this can be exploited by the adversary. We bound the probability of such a collision ($Y'^{E[X']} = Y^{E[X]}$) of honest shares X, Y, X', Y' in Lemma 5.1. Furthermore, notice when reducing to $Gsqdh$, we wish to transfer the distinguishing power of the adversary in the $Gsodhro$ game to $Gsqdh$. However, the adversary in $Gsodhro$ might also have and advantage without calling the random oracle on the desired Diffie-Hellman secret. To see this, imagine the adversary queries $\mathsf{XTR}(X, Y, salt)$ and $\mathsf{XTR}(X', Y', salt)$ for honest X, Y, X' but dishonest Y' such that $Y'^{E[X']} = Y^{E[X]}$. (i.e. X, Y has the same DH secret as X', Y' but Y' is under adversarial control.) In the ideal game, outputs of these calls are different with high probability. We eliminate this case by reducing to game $Gccoll^{b,grp}$. □

Formal proof of Lemma 5.6. We bound the advantage of an adversary interacting with game $Gsodhro^{b,grp}$ through a game-hopping argument. We define the pseudocode of hybrid games $G_i^{b,grp}$ for $i \in [11]$ in Figures 40 to 43. Observe that $G_1^{b,grp} \overset{code}{\equiv} Gsodhro^{b,grp}$. We also define pseudocode of reductions $\mathcal{R}_{hcoll}^{b,grp}$, $\mathcal{R}_{ccoll}^{b,grp}$, $\mathcal{R}_{stat}^{b,grp}$, and $\mathcal{R}_{alpha}^{b,grp}$ in Figures 35, 36, 37, and 38, respectively. Informally, we prove the following chain of game hops: (game parameters are not shown)

$$G_1 \overset{code}{\equiv} G_2 \overset{comp}{\approx} G_3 \overset{code}{\equiv} G_4 \overset{code}{\equiv} G_5 \overset{code}{\equiv} G_6 \overset{comp}{\approx} G_7 \overset{code}{\equiv} G_8 \overset{stat}{\approx} G_9 \overset{code}{\equiv} G_{10} \overset{comp}{\approx} G_{11}$$

To formally prove the chain of the game hops, we need the following lemma for computational and statistical game hops:

Lemma 5.7. *Let $\mathcal{A}$ be an adversary and for $b \in \{0, 1\}$, $G_0^b \overset{code}{\equiv} \mathcal{R}^b \to H^0$ and $G_1^b \overset{code}{\equiv} \mathcal{R}^b \to H^1$ for games G_0^b, G_1^b, H^0, H^1 and reduction $\mathcal{R}^b$. Then,*

$$\mathsf{Adv}(\mathcal{A}, G_0^b) \leq \mathsf{Adv}(\mathcal{A}, G_1^b) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^0, H^b) + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^1, H^b).$$

Proof.

$$\begin{aligned}
\mathsf{Adv}(\mathcal{A}, G_0^b) &= |\Pr[1 = \mathcal{A} \to (\mathcal{R}^0 \to H^0)] - \Pr[1 = \mathcal{A} \to (\mathcal{R}^1 \to H^0)]| \\
&= |\Pr[1 = (\mathcal{A} \to \mathcal{R}^0) \to H^0] - \Pr[1 = (\mathcal{A} \to \mathcal{R}^0) \to H^1] \\
&\quad + \Pr[1 = \mathcal{A} \to (\mathcal{R}^0 \to H^1)] - \Pr[1 = \mathcal{A} \to (\mathcal{R}^1 \to H^1)] \\
&\quad + \Pr[1 = (\mathcal{A} \to \mathcal{R}^1) \to H^1] - \Pr[1 = (\mathcal{A} \to \mathcal{R}^1) \to H^0]| \\
&\leq \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^0, H^b) \\
&\quad + \mathsf{Adv}(\mathcal{A}, \mathcal{R}^b \to H^1) \\
&\quad + \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^1, H^b).
\end{aligned}$$

□

Next, we prove the following code equivalences used by computational and statistical game hops.

Claim 5.8 (Reductions)**.**

$$
\begin{aligned}
G_2^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{hcoll}^{b,grp} \rightarrow Ghcoll^{0,grp},\\
G_3^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{hcoll}^{b,grp} \rightarrow Ghcoll^{1,grp},\\
G_6^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{ccoll}^{b,grp} \rightarrow Gccoll^{0,grp},\\
G_7^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{ccoll}^{b,grp} \rightarrow Gccoll^{1,grp},\\
G_8^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{stat}^{b,grp} \rightarrow Gstat^{0,grp},\\
G_9^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{stat}^{b,grp} \rightarrow Gstat^{1,grp},\\
G_{10}^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{alpha}^{b,grp} \rightarrow Galpha^{0,grp},\\
G_{11}^{b,grp} &\overset{code}{\equiv} \mathcal{R}_{alpha}^{b,grp} \rightarrow Galpha^{1,grp}.
\end{aligned}
$$

Proof. All code equivalences can be simply proved by inlining the code of oracles of the games $Ghcoll^{b,grp}$, $Gccoll^{b,grp}$, $Gstat^{b,grp}$, and $Galpha^{b,grp}$ in the reductions. □

In the following, we elaborate on changes in each game hop. In Figure 40 to 43, we might remove redundant assertions or variable assignments as a result of inlining. We also use **foreach** $(X', Y', salt)$ **in** T' for all tables T' instead of **foreach** $(X', Y', salt')$ **in** T' : **if** $salt' = salt$: where $salt$ is given as an argument. (i.e. we lookup for entries in the table with the given $salt$.) We use red lines to indicate changes from previous game. We use cyan, violet, brown, and green lines to show correspondence between adjacent columns.

Claim 5.9. $G_1 \overset{code}{\equiv} G_2$.

Proof. From G_1 to G_2, we replace $S[h, salt]$ with $S[X, Y, salt]$ and before sampling a new value for $S[X, Y, salt]$, check whether $S[X, Y, salt]$ or $S[Y, X, salt]$ has been already set before. This is one implementation for semantics of $\mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$. □

Claim 5.10. $G_2 \overset{comp}{\approx} G_3$.

Proof. From G_2 to G_3, we reduce to indistinguishability of game $Ghcoll^{b,grp}$ (honest collisions) with reduction $\mathcal{R}_{hcoll}^{b,grp}$. Advantage of an adversary in game $Ghcoll^{b,grp}$ is bounded in Lemma 5.1. □

Claim 5.11. $G_3 \overset{code}{\equiv} G_4$.

Proof. From G_3 to G_4, we merge the **foreach** loop and the following **if** conditions into one loop by dropping the codition $\{X, Y\} \neq \{X', Y'\}$. □

Claim 5.12. $G_4 \overset{code}{\equiv} G_5$.

Proof. From G_4 to G_5, we inline the code of H in XTR. □

Claim 5.13. $G_5 \overset{code}{\equiv} G_6$.

Proof. From G_5 to G_6, we replace the functionality provided by a common table T between the random oracle H and XTR with two tables T_{H} and T_{XTR} which are viewpoints of each oracle about hash values of group members (DH secrets) and salts. They store hash values corresponding to their inputs. It is required that the behaviour of two tables be consistent with one table T. This is achieved by iterating over values in the other table (T_{XTR} in oracle H and T_{H} in oracle XTR) and returning appropriate value upon a related input. The consistency of calls to XTR is acheived by iterating over T_{XTR} and checking for a collision $Y'^{E[X']} = Y^{E[X]}$. The consistency of calls to H is ensured by looking up the input in T_{H} in the very beginning of code of oracle H.

We formally prove and verify the code equivalence in SSBee by separating the common table T logic and reducing G_5 and G_6 with reduction $\mathcal{R}_{56}$ to G'_5 and G'_6 such that $G_5 \overset{code}{\equiv} \mathcal{R}_{56} \rightarrow G'_5$ and $G_6 \overset{code}{\equiv} \mathcal{R}_{56} \rightarrow G'_6$ and formally verifying code equivalence of G'_5 and G'_6 in SSBee. Notice that in the code of G'_6, looking up $Y^{E[X]}$ in T_{H} is the same as iterating over the table to find a matching entry. (Since SSBee does not support iteration control statements, we reduce the loops to absolutely necessary ones.) We discuss the required invariants in Lemma 5.20. Refer to Figure 44 for code of reduction $\mathcal{R}_{56}$ and games G'_5 and G'_6.

Observe that in the ideal game (i.e. $G_6^{b=1}$), S stores hash values for only honest Y's and T_{XTR} stores hash values for only dishonest Y's. Therefore, the two loops for looking for possible collisions of the form $Y'^{E[X']} = Y^{E[X]}$, find collisions between either only honest shares or only dishonest shares. However, observe that the reduction $\mathcal{R}^{grp}$ checks for all collisions (including possibly those between honest and dishonest shares) in the very beginning because the sets S and T_{XTR} are merged and T_{XTR} contains both honest and dishonest shares. Hence, we introduce table L to enable us to check for all collisions before merging S and T_{XTR}. This requires us to reduce to game $Gccoll^{b,grp}$ (cross collisions). □

Claim 5.14. $G_6 \overset{comp}{\approx} G_7$.

Proof. From G_6 to G_7, we reduce to indistinguishability of $Gccoll^{b,grp}$ with reduction $\mathcal{R}_{ccoll}^{b,grp}$ and prevent all cross collisions in the beginning. Notice that the **foreach** loop in the code of oracle FIND provided by the game checks for collisions only between honest and dishonest shares with the additional condition $\big((E_Y = \perp) \neq (E_{Y'} = \perp)\big)$. This conditions states that honesty of Y and Y' shall not be the same or in other words, exactly one of Y or Y' should be dishonest. Therefore, in the game $G_7^{b,grp}$ where $Gccoll^{1,grp}$ is idealized, all possible collisions among honest shares only, dishonest shares only, and between honest and dishonest shares are captured with these three **foreach** loops which can be all merged together in one loop in $G_8^{b,grp}$. □

Claim 5.15. $G_7 \overset{code}{\equiv} G_8$.

Proof. From G_7 to G_8, we merge tables S and T_{XTR} and reuse T_{XTR} for lookups instead of L. This requires taking into account the game parameter b in the code of oracle

H. We shall not return the same hash value as sampled by XTR oracle in the ideal games for honest Y's. We also merge the **foreach** loops that lookup values in S, L, and T_{XTR}. Observe that this makes a difference between idealized games $G_7^{1,grp}$ and $G_8^{1,grp}$ while the real games $G_7^{0,grp}$ and $G_8^{0,grp}$ are excatly the same as all shares are stored in T_{XTR} and the loop over this table captures all collisions. □

Claim 5.16. $G_8 \overset{stat}{\approx} G_9$.

Proof. From G_8 to G_9, we reduce to indistinguishability of game $Gstat^{b,grp}$ with reduction $\mathcal{R}_{stat}^{b,grp}$. Advantage of any adversary in game $Gstat^{b,grp}$ is 0 as proved in Lemma 5.4. □

Claim 5.17. $G_9 \overset{code}{\equiv} G_{10}$.

Proof. From G_9 to G_{10}, we replace $E[X] \leftarrow \alpha x$ with $E[X] \leftarrow x$ and update operations. □

Claim 5.18. $G_{10} \overset{code}{\approx} G_{11}$.

Proof. From G_{10} to G_{11}, we reduce to indistinguishability of game $Galpha^{b,grp}$ and replace $(Y'^{E[X']})^{\alpha} = (Y^{E[X]})^{\alpha}$ with $Y'^{E[X']} = Y^{E[X]}$ while adding an assertion after sampling α to avoid trivial value $\alpha = q$. Advantage of an adversary in game $Galpha^{b,grp}$ is bounded in Lemma 5.3. □

Claim 5.19. $G_{11}^{b,grp} \overset{code}{\equiv} \mathcal{R}^{grp} \rightarrow Gsqdh^{b,grp}$.

Proof. Finally, game G_{11} is exactly the composition of reduction $\mathcal{R}^{grp}$ with $Gsqdh^{b,grp}$. We use the oracles provided by game $Gsqdh^{b,grp}$. See colored lines in Figure 43. Notice b can be safely replaced with $\mathsf{CHECK}(Z^{E[X]^{-1}E[Y]^{-1}})$ oracle because at that point we know $Z = Y^{\alpha E[X]} = g^{E[X]E[Y]\alpha^2}$, considering short circuit semantics of the **if** condition evaluation. □

In the following, we abuse the notation and, for example, write $2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp})$ instead of $\mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{0,grp}, Ghcoll^{b,grp}) + \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{1,grp}, Ghcoll^{b,grp})$.

Now by applying lemma 5.7 to the code equivalences in Claim 5.8 and 5.19, we get

$$
\begin{aligned}
\mathsf{Adv}(\mathcal{A}, G_0^{b,grp}) &= \mathsf{Adv}(\mathcal{A}, G_2^{b,grp}) \\
&\leq \mathsf{Adv}(\mathcal{A}, G_3^{b,grp}) \\
&\quad + \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{0,grp}, Ghcoll^{b,grp}) + \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{1,grp}, Ghcoll^{b,grp}) \\
&= \mathsf{Adv}(\mathcal{A}, G_6^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) \\
&\leq \mathsf{Adv}(\mathcal{A}, G_7^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{ccoll}^{b,grp}, Gccoll^{b,grp}) \\
&= \mathsf{Adv}(\mathcal{A}, G_8^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{ccoll}^{b,grp}, Gccoll^{b,grp}) \\
&= \mathsf{Adv}(\mathcal{A}, G_9^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{ccoll}^{b,grp}, Gccoll^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{stat}^{b,grp}, Gstat^{b,grp}) \\
&= \mathsf{Adv}(\mathcal{A}, G_{10}^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{ccoll}^{b,grp}, Gccoll^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{stat}^{b,grp}, Gstat^{b,grp}) \\
&= \mathsf{Adv}(\mathcal{A}, G_{11}^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{ccoll}^{b,grp}, Gccoll^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{stat}^{b,grp}, Gstat^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{alpha}^{b,grp}, Galpha^{b,grp}) \\
&= \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}^{grp}, Gsqdh^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{hcoll}^{b,grp}, Ghcoll^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{ccoll}^{b,grp}, Gccoll^{b,grp}) \\
&\quad + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{stat}^{b,grp}, Gstat^{b,grp}) + 2 \times \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}_{alpha}^{b,grp}, Galpha^{b,grp}).
\end{aligned}
$$

□

Lemma 5.20. $G'_5 \overset{code}{\equiv} G'_6$. *(verified in SSBee)*

Proof. In the following, we introduce three state relations (observations) as necessary conditions for equivalence, which turn out to be sufficient too. Then, as discussed in Section 2.1 and 2.5, we formally prove code equivalence of the games, using an induction over the oracle calls to show three properties for each oracle. Firstly, on the same inputs to corresponding oracles from left (G'_5) and right (G'_6) games, our state relations are preserved across the calls and, hence, are invariants. Secondly, again on the same inputs, either both of the oracles abort or neither abort. Thirdly, on the same inputs, if the oracles do not abort, they return the same output.

$G_1^{b,grp}$

Parameters

b : idealization bit

grp : group description

State

E : table

S : table

T : table

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$

$q \leftarrow \mathsf{ord}(g)$

$x \leftarrow\!\!\$\ Z_q$

$X \leftarrow g^x$

$E[X] \leftarrow x$

return X

XTR$(X, Y, salt)$

assert $E[X] \neq \perp \wedge$

$\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$

$\mathsf{alg}(salt) \in \mathcal{H}$

$alg \leftarrow \mathsf{alg}(salt)$

if $b \wedge E[Y] \neq \perp$:

$h \leftarrow \mathsf{dh}\langle \mathsf{sort}(X, Y)\rangle$

if $S[h, salt] = \perp$:

$S[h, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $S[h, salt]$

return $\mathsf{H}(Y^{E[X]}, salt)$

H$(Z, salt)$

if $T[Z, salt] \neq \perp$:

return $T[Z, salt]$

$alg \leftarrow \mathsf{alg}(salt)$

$T[Z, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $T[Z, salt]$

$G_2^{b,grp}$

Parameters

same as the left

State

same as the left

DHGEN()

same as the left

XTR$(X, Y, salt)$

assert $E[X] \neq \perp \wedge$

$\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$

$\mathsf{alg}(salt) \in \mathcal{H}$

$alg \leftarrow \mathsf{alg}(salt)$

if $b \wedge E[Y] \neq \perp$:

if $S[X, Y, salt] \neq \perp$:

return $S[X, Y, salt]$

if $S[Y, X, salt] \neq \perp$:

return $S[Y, X, salt]$

$S[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $S[X, Y, salt]$

return $\mathsf{H}(Y^{E[X]}, salt)$

H$(Z, salt)$

same as the left

$G_3^{b,grp}$

Parameters

same as the left

State

same as the left

DHGEN()

same as the left

XTR$(X, Y, salt)$

assert $E[X] \neq \perp \wedge$

$\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$

$\mathsf{alg}(salt) \in \mathcal{H}$

$alg \leftarrow \mathsf{alg}(salt)$

if $b \wedge E[Y] \neq \perp$:

foreach $(X', Y', salt)$ **in** S :

if $Y'^{E[X']} = Y^{E[X]} \wedge$

$\{X, Y\} \neq \{X', Y'\}$:

return $S[X', Y', salt]$

if $S[X, Y, salt] \neq \perp$:

return $S[X, Y, salt]$

if $S[Y, X, salt] \neq \perp$:

return $S[Y, X, salt]$

$S[X, Y, salt] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{len}(alg)}$

return $S[X, Y, salt]$

return $\mathsf{H}(Y^{E[X]}, salt)$

H$(Z, salt)$

same as the left

Figure 40: Games $G_1^{b,grp}$, $G_2^{b,grp}$, $G_3^{b,grp}$

$G_4^{b,grp}$

Parameters

same as $G_3^{b,grp}$

State

same as $G_3^{b,grp}$

DHGEN()

same as $G_3^{b,grp}$

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
$alg \leftarrow \mathsf{alg}(salt)$
if $b \wedge E[Y] \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]}$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
  return $S[X, Y, salt]$
return $\mathsf{H}(Y^{E[X]}, salt)$

H$(Z, salt)$

/ no change, repeated for comparison
if $T[Z, salt] \neq \bot$:
  return $T[Z, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T[Z, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
return $T[Z, salt]$

$G_5^{b,grp}$

Parameters

same as the left

State

/ no change, repeated for comparison
E : table
S : table
T : table

DHGEN()

same as the left

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
$alg \leftarrow \mathsf{alg}(salt)$
if $b \wedge E[Y] \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]}$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
  return $S[X, Y, salt]$
if $T[Y^{E[X]}, salt] \neq \bot$:
  return $T[Y^{E[X]}, salt]$
$T[Y^{E[X]}, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
return $T[Y^{E[X]}, salt]$

H$(Z, salt)$

same as the left

$G_6^{b,grp}$

State

E : table
S : table
T_{H} : table
T_{XTR} : table

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
$alg \leftarrow \mathsf{alg}(salt)$
if $b \wedge E[Y] \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]}$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
  return $S[X, Y, salt]$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
foreach $(Z, salt)$ **in** T_{H} :
  if $Y^{E[X]} = Z$:
    return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $Y^{E[X]} = Z$:
    return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\$ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

Figure 41: Games $G_4^{b,grp}$, $G_5^{b,grp}$, $G_6^{b,grp}$

$G_7^{b,grp}$

State

E : table

S : table

L : table

T_{H} : table

T_{XTR} : table

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** L :
  if $Y'^{E[X']} = Y^{E[X]} \wedge$
    $\big((E[Y] = \bot) \neq (E[Y'] = \bot)\big)$:
    return $L[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
if $b \wedge E[Y] \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{E[X']} = Y^{E[X]}$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
  $L[X, Y, salt] \leftarrow S[X, Y, salt]$
  return $L[X, Y, salt]$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
foreach $(Z, salt)$ **in** T_{H} :
  if $Y^{E[X]} = Z$:
    return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
$L[X, Y, salt] \leftarrow T_{\mathsf{XTR}}[X, Y, salt]$
return $L[X, Y, salt]$

H$(Z, salt)$

same as $G_6^{b,grp}$

$G_8^{b,grp}$

State

E : table

T_{H} : table

T_{XTR} : table

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(Z, salt)$ **in** T_{H} :
  if $Y^{E[X]} = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $Y^{E[X]} = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

$G_9^{b,grp}$

State

E : table

T_{H} : table

T_{XTR} : table

α : integer

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
if $\alpha = \bot$:
  $\alpha \leftarrow\!\$\ Z_q$
$x \leftarrow\!\$\ Z_q$
$X \leftarrow g^{\alpha x}$
$E[X] \leftarrow \alpha x$
return X

XTR$(X, Y, salt)$

same as the left

H$(Z, salt)$

same as the left

Figure 42: Games $G_7^{b,grp}$, $G_8^{b,grp}$, $G_9^{b,grp}$

$G_{10}^{b,grp}$

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
if $\alpha = \bot$:
  $\alpha \leftarrow\!\$\ Z_q$
$x \leftarrow\!\$\ Z_q$
$X \leftarrow (g^x)^\alpha$
$E[X] \leftarrow x$
return X

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $(Y'^{E[X']})^\alpha = (Y^{E[X]})^\alpha$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(Z, salt)$ **in** T_{H} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

$G_{11}^{b,grp}$

State

E : table
T_{H} : table
T_{XTR} : table
α : integer

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
if $\alpha = \bot$:
  $\alpha \leftarrow\!\$\ Z_q$
assert $\alpha \neq q$
$x \leftarrow\!\$\ Z_q$
$X \leftarrow (g^\alpha)^x$
$E[X] \leftarrow x$
return X

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(Z, salt)$ **in** T_{H} :
  if $(Y^{E[X]})^\alpha = Z$:
    if $E[Y] = \bot \vee \neg b$:
      return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

same as the left

$\mathcal{R}^{grp}$

DHGEN()

$g \leftarrow \mathsf{gen}(grp)$
$q \leftarrow \mathsf{ord}(g)$
$A \leftarrow\!\$\ \mathsf{SAMPLE}()$
assert $A \neq g^q$
$x \leftarrow\!\$\ Z_q$
$X \leftarrow A^x$
$E[X] \leftarrow x$
return X

XTR$(X, Y, salt)$

assert $E[X] \neq \bot \wedge$
  $\mathsf{grp}(X) = \mathsf{grp}(Y) = grp \wedge$
  $\mathsf{alg}(salt) \in \mathcal{H}$
foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(Z, salt)$ **in** T_{H} :
  if $\mathsf{DDH}(Y^{E[X]}, Z)$:
    if $E[Y] = \bot \vee$
      $\neg\mathsf{CHECK}(Z^{E[X]^{-1}E[Y]^{-1}})$:
      return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

H$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
$alg \leftarrow \mathsf{alg}(salt)$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $\mathsf{DDH}(Y^{E[X]}, Z)$:
    if $E[Y] = \bot \vee$
      $\neg\mathsf{CHECK}(Z^{E[X]^{-1}E[Y]^{-1}})$:
      return $T_{\mathsf{XTR}}[X, Y, salt]$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\$\ \{0,1\}^{\mathsf{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

Figure 43: Games $G_{10}^{b,grp}$, $G_{11}^{b,grp}$. Reduction $\mathcal{R}^{grp}$ is repeated for comparison.

$\mathcal{R}_{56}^{b,grp}$

State

S : table

DHGEN()

return DHGEN()

XTR$(X, Y, salt)$

assert DHGET$(X) \neq \bot \wedge$
  grp(X) = grp$(Y) = grp \wedge$
  alg$(salt) \in \mathcal{H}$
$alg \leftarrow$ alg$(salt)$
if $b \wedge$ DHGET$(Y) \neq \bot$:
  foreach $(X', Y', salt)$ **in** S :
    if $Y'^{\text{DHGET}(X')} = Y^{\text{DHGET}(X)}$:
      return $S[X', Y', salt]$
  $S[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\text{len}(alg)}$
  return $S[X, Y, salt]$
return TXTR$(X, Y, salt)$

H$(Z, salt)$

return TH$(Z, salt)$

G'_5

Parameters

grp : group description

State

T : table
E : table

DHGEN()

$g \leftarrow$ gen(grp)
$q \leftarrow$ ord(g)
$x \leftarrow\!\$\ Z_q$
$X \leftarrow g^x$
$E[X] \leftarrow x$
return X

DHGET(X)

return $E[X]$

TXTR$(X, Y, salt)$

$Z \leftarrow Y^{E[X]}$
if $T[Z, salt] \neq \bot$:
  return $T[Z, salt]$
$alg \leftarrow$ alg$(salt)$
$T[Z, salt] \leftarrow\!\$\ \{0,1\}^{\text{len}(alg)}$
return $T[Z, salt]$

TH$(Z, salt)$

if $T[Z, salt] \neq \bot$:
  return $T[Z, salt]$
$alg \leftarrow$ alg$(salt)$
$T[Z, salt] \leftarrow\!\$\ \{0,1\}^{\text{len}(alg)}$
return $T[Z, salt]$

G'_6

Parameters

grp : group description

State

T_{H} : table
T_{XTR} : table
E : table

DHGEN()

same as the left

DHGET(X)

same as the left

TXTR$(X, Y, salt)$

foreach $(X', Y', salt)$ **in** T_{XTR} :
  if $Y'^{E[X']} = Y^{E[X]}$:
    return $T_{\mathsf{XTR}}[X', Y', salt]$
if $T_{\mathsf{H}}[Y^{E[X]}, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
$T_{\mathsf{XTR}}[X, Y, salt] \leftarrow\!\$\ \{0,1\}^{\text{len}(alg)}$
return $T_{\mathsf{XTR}}[X, Y, salt]$

TH$(Z, salt)$

if $T_{\mathsf{H}}[Z, salt] \neq \bot$:
  return $T_{\mathsf{H}}[Z, salt]$
foreach $(X, Y, salt)$ **in** T_{XTR} :
  if $Y^{E[X]} = Z$:
    return $T_{\mathsf{XTR}}[X, Y, salt]$
$alg \leftarrow$ alg$(salt)$
$T_{\mathsf{H}}[Z, salt] \leftarrow\!\$\ \{0,1\}^{\text{len}(alg)}$
return $T_{\mathsf{H}}[Z, salt]$

Figure 44: Games G'_5 and G'_6, reduction $\mathcal{R}_{56}^{b,grp}$

Intuitively, the state relations are as follows:

$$E := E_{G'_5} = E_{G'_6}.$$
$$\forall X, Y, s, h\ \left(T_{\mathsf{XTR}}[X, Y, s] = h \neq \bot \Rightarrow T[Y^{E[X]}, s] = h\right).$$
$$\forall Z, s, h\ \left(T_{\mathsf{H}}[Z, s] = h \neq \bot \Rightarrow T[Z, s] = h\right).$$
$$\forall Z, s, h\ \Big(T[Z, s] = h \neq \bot \Rightarrow (T_{\mathsf{H}}[Z, s] = h \vee \exists X, Y.\ T_{\mathsf{XTR}}[X, Y, s] = h \wedge Y^{E[X]} = Z)\Big).$$

Namely, exponent tables $E_{G'_5}$ and $E_{G'_6}$ should be the same because of the same code for DHGEN oracle, which results in the first relation. We also expect that whenever an entry is inserted in tables $T_{\mathsf{XTR}}[X, Y, s]$ or $T_{\mathsf{H}}[Z, s]$ of game G'_6, there is a corresponding entry in table T of game G'_5, which gives us the second two relations. Conversely, for any entry inserted in table T of game G'_5, we expect there is a corresponding entry either in table $T_{\mathsf{XTR}}[X, Y, s]$ or $T_{\mathsf{H}}[Z, s]$ of game G'_6, which gives us the last relation.

We prove these state relations to be invariant and they turn out to be sufficient to prove the equivalence. However, since SSBee does not support any iteration control statements (i.e. loops), we encode the lookup loops in game G'_6 inspired by the epsilon operator in Hilbert's epsilon calculus [AZ24]. Precisely, we define an abstract function in SSBee that performs the lookup in table $T_{\mathsf{XTR}}[X, Y, s]$ to find a pair (X', Y') such that $Y^{E[X]} = Y'^{E[X']}$ or returns $\bot$ if there is no such pair in the table. We exactly specify these properties for the function and use it as a lemma for code equivalence proof obligations. Let the abstract function be ϵ. We require the following properties for ϵ:

$$\forall T', Z, s.\ \Big(\epsilon(T', Z, s) = \bot \Leftrightarrow (\forall X, Y.\ T'[X, Y, s] \neq \bot \Rightarrow Z \neq Y^{E[X]})\Big).$$
$$\forall T', Z, s.\ \Big((X, Y) := \epsilon(T', Z, s) \neq \bot \Rightarrow (Z = Y^{E[X]} \wedge T'[X, Y, s] \neq \bot)\Big).$$
$$\forall T', X, Y, s.\ \Big(T'[X, Y, s] \neq \bot \Rightarrow \epsilon(T', Y^{E[X]}, s) = (X, Y)\Big).$$

The first relation state a necessary and sufficient condition when ϵ returns $\bot$. (i.e. when no pair (X, Y) collides with Z) The second relation states a necessary condition for the output of ϵ that when ϵ returns a pair, it is indeed a collision and it exists in the table. The last relation states a sufficient condition for what is returned by ϵ. Notice that for the function to be well-defined, we need to show that for each group memebr Z and salt s, there can be at most one pair (X, Y) in the table such that $T'[X, Y, s] \neq \bot$ and $Y^{E[X]} = Z$. We refer to this property in our game as uniqueness. Observe that assuming ϵ is a mathematical function (which is the case for abstract functions in SSBee or SMT solvers), the uniqueness property is implicit and implied by the last relation. Uniqueness property is stated as follows:

$$\forall T', X, Y, s.\ \Big(T'[X, Y, s] \neq \bot \Rightarrow \big(\forall X', Y'.\ T'[X', Y', s] \neq \bot \wedge Y'^{E[X']} = Y^{E[X]} \Rightarrow X = X' \wedge Y = Y'\big)\Big)$$

We prove the uniqueness property in SSBee assuming that if ϵ returns $\perp$, there is no colliding pair in the table. (direction $\Rightarrow$ of the first relation) Notice that we do not assume (as we should not) the other direction (what is returned when there is at least one colliding pair) to prove the uniqueness. Another formulation is to allow ϵ to be a partial function and define the last relation as follows so that $\epsilon(T', Y^{E[X]}, s)$ is defined if there is exactly one colliding pair in the table. Then, uniqueness property simply states that the function is total.

$$\forall T', Z, s.\ \Big(\big(\exists! X, Y.\ T'[X, Y, s] \neq \perp \wedge Z = Y^{E[X]}\big) \Rightarrow \epsilon(T', Z, s) \neq \perp\Big).$$

However, we do not state the relation with uniqueness quantification (as above), but rather prove the uniqueness separately for our verification to be sound. Moreover, we leave the lemma about properties of ϵ without proof. Generally speaking, modeling a lookup loop as a determinitic function of table and lookup key requires specification of an iteration order or proving that the pair if exists is unique. In fact one can prove the lemma in a program verification tool that supports program logic for the loop. Therefore, using the uniqueness property as a precondition for the loop, one can prove the aforementioned properties in the tool.

Since we have automatized the code equivalence proof, we refer the reader to the code of packages `g5P` and `g6P` as well as proof file and invariants file in Listings 13 to 17. Refer to [Raj25d] for the full SSBee project. Notice that ϵ is called `find_collision_in_TXTR` in the code of package `g6P`. Also, we have refrained from defining oracle DHGEN as it is tangent to the core logic of the games. We, though, keep the invariant $E_{G'_5} = E_{G'_6}$ because it is necessary for proving obligations related to oracles TXTR and TH.

```
package g5P {
    state {
        E: Table(Bits(*), Integer),
        T: Table((Bits(*), Bits(*)), Bits(*)),
    }

    params {
        exp: fn Bits(*), Integer -> Bits(*)
    }

    oracle TXTR(X: Bits(*), Y: Bits(*), s: Bits(*)) -> Bits(*) {
        Z <- exp(Y, Unwrap(E[X]));
        if (T[(Z, s)] != None as Bits(*)) {
<           return Unwrap(T[(Z, s)]);
        }
        h <-$ Bits(*);
        T[(Z, s)] <- Some(h);
        return h;
    }

    oracle TH(Z: Bits(*), s: Bits(*)) -> Bits(*) {
        if (T[(Z, s)] != None as Bits(*)) {
            return Unwrap(T[(Z, s)]);
        }
```

```
        h <-$ Bits(*);
        T[(Z, s)] <- Some(h);
        return h;
    }
}
```

Listing 13: Code of package `g5P`

```
package g6P {
    state {
        E: Table(Bits(*), Integer),
        TH: Table((Bits(*), Bits(*)), Bits(*)),
        TXTR: Table((Bits(*), Bits(*), Bits(*)), Bits(*)),
    }

    params {
        exp: fn Bits(*), Integer -> Bits(*),
        find_collision_in_TXTR:
            fn Table((Bits(*), Bits(*), Bits(*)), Bits(*)),
               Bits(*),
               Bits(*) -> Maybe((Bits(*), Bits(*)))
    }

    oracle TXTR(X: Bits(*), Y: Bits(*), s: Bits(*)) -> Bits(*) {
        Z <- exp(Y, Unwrap(E[X]));
        if (TH[(Z, s)] != None as Bits(*)) {
            return Unwrap(TH[(Z, s)]);
        }
        XpYp <- find_collision_in_TXTR(TXTR, Z, s);
        if (XpYp != None as (Bits(*), Bits(*))) {
            (Xp, Yp) <- parse Unwrap(XpYp);
            return Unwrap(TXTR[(Xp, Yp, s)]);
        }
        h <-$ Bits(*);
        TXTR[(X, Y, s)] <- Some(h);
        return h;
    }

    oracle TH(Z: Bits(*), s: Bits(*)) -> Bits(*) {
        if (TH[(Z, s)] != None as Bits(*)) {
            return Unwrap(TH[(Z, s)]);
        }
        XY <- find_collision_in_TXTR(TXTR, Z, s);
        if (XY != None as (Bits(*), Bits(*))) {
            (X, Y) <- parse Unwrap(XY);
            return Unwrap(TXTR[(X, Y, s)]);
        }
        h <-$ Bits(*);
        TH[(Z, s)] <- Some(h);
        return h;
    }
}
```

Listing 14: Code of package `g6P`

```
composition G5 {
    const exp: fn Bits(*), Integer -> Bits(*);

    instance g5 = g5P {
        params {
            exp: exp
        }
    }

    compose {
        adversary: {
            TH: g5,
            TXTR: g5
        }
    }
}
composition G6 {
    const exp: fn Bits(*), Integer -> Bits(*);
    const find_collision_in_TXTR:
            fn Table((Bits(*), Bits(*), Bits(*)), Bits(*)),
               Bits(*),
               Bits(*) -> Maybe((Bits(*), Bits(*)));

    instance g6 = g6P {
        params {
            exp: exp,
            find_collision_in_TXTR: find_collision_in_TXTR
        }
    }

    compose {
        adversary: {
            TH: g6,
            TXTR: g6
        }
    }
}
```

Listing 15: Definition of games (compositions) G'_5 and G'_6 in SSBee

```
proof Proof {
    const exp: fn Bits(*), Integer -> Bits(*);
    const find_collision_in_TXTR:
            fn Table((Bits(*), Bits(*), Bits(*)), Bits(*)),
               Bits(*),
               Bits(*) -> Maybe((Bits(*), Bits(*)));

    instance g5 = G5 {
        params {
            exp: exp
        }
    }

    instance g6 = G6 {
```

```
        params {
            exp: exp,
            find_collision_in_TXTR: find_collision_in_TXTR
        }
    }

    gamehops {
        equivalence g5 g6 {
            TH : {
                invariant: [
                    ./proofs/invariant.smt2
                ]

                lemmas {
                    same-output:  [no-abort, lemma-find-collision]
                    invariant:    [no-abort, lemma-find-collision]
                    equal-aborts: [lemma-find-collision]
                    assert-uniqueness: [no-abort, assume-uniqueness-and-none-collision]
                }
            }
            TXTR : {
                invariant: [
                    ./proofs/invariant.smt2
                ]

                lemmas {
                    invariant:    [no-abort, lemma-find-collision]
                    same-output:  [no-abort, lemma-find-collision]
                    equal-aborts: [lemma-find-collision]
                    assert-uniqueness: [no-abort, assume-uniqueness-and-none-collision]
                }
            }
        }
    }
}
```

Listing 16: Code of proof file

```
(define-fun invariant-TH-implies-T
    (
        (E (Array Bits_* (Maybe Int)))
        (T (Array (Tuple2 Bits_* Bits_*) (Maybe Bits_*)))
        (TH (Array (Tuple2 Bits_* Bits_*) (Maybe Bits_*)))
        (TXTR (Array (Tuple3 Bits_* Bits_* Bits_*) (Maybe Bits_*)))
    )
    Bool
    ; TH[Z, s] = h != None => T[Z, s] = h
    (forall
        (
            (Z Bits_*)
            (s Bits_*)
        )
        (let
            (
```

```
                (h (select TH (mk-tuple2 Z s)))
            )
            (=>
                (not ((_ is mk-none) h))
                (= h (select T (mk-tuple2 Z s)))
))))
(define-fun invariant-T-NotNone-implies-TH-and-TXTR
    (
        (E (Array Bits_* (Maybe Int)))
        (T (Array (Tuple2 Bits_* Bits_*) (Maybe Bits_*)))
        (TH (Array (Tuple2 Bits_* Bits_*) (Maybe Bits_*)))
        (TXTR (Array (Tuple3 Bits_* Bits_* Bits_*) (Maybe Bits_*)))
    )
    Bool
    ; T[Z, s] = h != None => TH[Z, s] = h or
    ; there exists X, Y such that Y^E[X] = Z and TXTR[X, Y, s] = h
    (forall
        (
            (Z Bits_*)
            (s Bits_*)
        )
        (let
            (
                (h (select T (mk-tuple2 Z s)))
            )
            (=>
                (not ((_ is mk-none) h))
                (or
                    (= h (select TH (mk-tuple2 Z s)))
                    (exists
                        (
                            (X Bits_*)
                            (Y Bits_*)
                        )
                        (and
                            (= h (select TXTR (mk-tuple3 X Y s)))
                            (= Z (<<func-exp>> Y (maybe-get (select E X))))
)))))))
```

Listing 17: Invariant file

```
(define-fun invariant-TXTR-implies-T
    (
        (E (Array Bits_* (Maybe Int)))
        (T (Array (Tuple2 Bits_* Bits_*) (Maybe Bits_*)))
        (TH (Array (Tuple2 Bits_* Bits_*) (Maybe Bits_*)))
        (TXTR (Array (Tuple3 Bits_* Bits_* Bits_*) (Maybe Bits_*)))
    )
    Bool
    ; TXTR[X, Y, s] = h != None => T[Y^E[X], s] = h
    (forall
        (
            (X Bits_*)
            (Y Bits_*)
```

```
            (s Bits_*)
        )
        (let
            (
                (h (select TXTR (mk-tuple3 X Y s)))
            )
            (=>
                (not ((_ is mk-none) h))
                (= h (select T (mk-tuple2 (<<func-exp>> Y (maybe-get (select E X))) s)))
            )
        )
    )
)
(define-fun invariant
    (
        (state-g5  <GameState_G5_<$$>>)
        (state-g6  <GameState_G6_<$$>>)
    )
    Bool
    (let
        (
            (E_left (<pkg-state-g5P-<$$>-E> (<game-G5-<$$>-pkgstate-g5> state-g5)))
            (E_right (<pkg-state-g6P-<$$>-E> (<game-G6-<$$>-pkgstate-g6> state-g6)))
            (T (<pkg-state-g5P-<$$>-T> (<game-G5-<$$>-pkgstate-g5> state-g5)))
            (TH (<pkg-state-g6P-<$$>-TH> (<game-G6-<$$>-pkgstate-g6> state-g6)))
            (TXTR (<pkg-state-g6P-<$$>-TXTR> (<game-G6-<$$>-pkgstate-g6> state-g6)))
        )
        (and
            (= E_left E_right)
            ; TH[Z, s] = h != None => T[Z, s] = h
            (invariant-TH-implies-T E_left T TH TXTR)
            ; T[Z, s] = h != None => TH[Z, s] = h or
            ; there exists X, Y such that Y^E[X] = Z and TXTR[X, Y, s] = h
            (invariant-T-NotNone-implies-TH-and-TXTR E_left T TH TXTR)
            ; TXTR[X, Y, s] = h != None => T[Y^E[X], s] = h
            (invariant-TXTR-implies-T E_left T TH TXTR)
        )
    )
)
```

Listing 18: Invariant file

```
(define-fun <relation-lemma-find-collision-g5-g6-TXTR>
    (and
        ; find(table, Z, s) = None <=> forall X,Y. table[X, Y, s] != None => Z != Y^E[X]
        (forall
            (
                (Zp Bits_*)
                (sp Bits_*)
                (table (Array (Tuple3 Bits_* Bits_* Bits_*) (Maybe Bits_*)))
            )
            (=
                ((_ is mk-none) (<<func-find_collision_in_TXTR>> table Zp sp))
                (forall
```

```
                    (
                        (Xp Bits_*)
                        (Yp Bits_*)
                    )
                    (=>
                        (not ((_ is mk-none) (select table (mk-tuple3 Xp Yp sp))))
                        (not (= Zp (<<func-exp>> Yp (maybe-get (select E Xp)))))
        ))))
        (forall
            (
                (Zp Bits_*)
                (Xp Bits_*)
                (Yp Bits_*)
                (sp Bits_*)
                (table (Array (Tuple3 Bits_* Bits_* Bits_*) (Maybe Bits_*)))
            )
            (and
                ; e = find(table, Z, s) => e_Y ^E[e_X] = Z and table[e_x, e_y, s] != None
                (let
                    (
                        (e (<<func-find_collision_in_TXTR>> table Zp sp))
                    )
                    (=>
                        (not ((_ is mk-none) e))
                        (let
                            (
                                (eX (el2-1 (maybe-get e)))
                                (eY (el2-2 (maybe-get e)))
                            )
                            (and
                               (not ((_ is mk-none) (select table (mk-tuple3 eX eY sp))))
                               (= Zp (<<func-exp>> eY (maybe-get (select E eX))))
                ))))
                ; table[X, Y, s] != None => find(table, y^E[X], s) = (X, Y)
                (=>
                    (not ((_ is mk-none) (select table (mk-tuple3 Xp Yp sp))))
                    (= (mk-some (mk-tuple2 Xp Yp))  (<<func-find_collision_in_TXTR>>
    table (<<func-exp>> Yp (maybe-get (select E Xp))) sp))
)))))
```

Listing 19: Lemma about ϵ operator (arguments of function are removed to fit the page)

```
(define-fun <relation-assume-uniqueness-and-none-collision-g5-g6-TXTR>
  (
    (old-state-g6  <GameState_G6_<$$>>)
    (return-g5 <OracleReturn-G5-<$$>-g5P-<$$>-TXTR>)
  )
  Bool
  (let
    ((state-g5 (<oracle-return-G5-<$$>-g5P-<$$>-TXTR-game-state> return-g5)))
    (let
        (
          (E_left (<pkg-state-g5P-<$$>-E> (<game-G5-<$$>-pkgstate-g5> state-g5)))
```

```
          (oldTXTR (<pkg-state-g6P-<$$>-TXTR> (<game-G6-<$$>-pkgstate-g6> old-state-g6)))
    )
        (and
          ; find(table, Z, s) = None => forall X,Y. table[X, Y, s] != None => Z != Y^E[X]
          (forall
            (
              (Zp Bits_*)
              (sp Bits_*)
              (table (Array (Tuple3 Bits_* Bits_* Bits_*) (Maybe Bits_*))))
            (=>
              ((_ is mk-none) (<<func-find_collision_in_TXTR>> table Zp sp))
              (forall
                (
                  (Xp Bits_*)
                  (Yp Bits_*)
                )
                (=>
                  (not ((_ is mk-none) (select table (mk-tuple3 Xp Yp sp))))
                  (not (= Zp (<<func-exp>> Yp (maybe-get (select E_left Xp)))))
              ))))
              ; TXTR[X, Y, s] != None =>
              ; forall X', Y' if TXTR[X', Y', s] != None and Y^E[X] = Y'^E[X'] =>
              ; X = S' and Y = Y'
              (forall
                (
                  (X Bits_*)
                  (Y Bits_*)
                  (s Bits_*)
                )
                (=>
                  (not ((_ is mk-none) (select oldTXTR (mk-tuple3 X Y s))))
                  (forall
                    (
                      (Xp Bits_*)
                      (Yp Bits_*)
                    )
                    (=>
                      (and
                        (not ((_ is mk-none) (select oldTXTR (mk-tuple3 Xp Yp s))))
                        (= (<<func-exp>> Y (maybe-get (select E_left X))) (<<func-exp>> Yp
    (maybe-get (select E_left Xp)))))
                      (and
                        (= X Xp)
                        (= Y Yp)

)))))))))
```

Listing 20: Uniqueness property assuming there is no collision in table if ϵ returns $\perp$

□

5.2.1 Proof of Lemma 5.1

Proof of Lemma 5.1. An adversary can distinguish $Ghcoll^{0,grp}$ from $Ghcoll^{1,grp}$ if upon a call to $\mathsf{FIND}(X, Y, salt)$, it receives $S[X', Y', salt]$ (from the loop), where $Y'^{E[X']} = Y^{E[X]}$ and $\{X, Y\} \neq \{X', Y'\}$, instead of $\perp$. However, X, Y, X', Y' are all honest and uniformly chosen by the game although the adversary can see the private exponents from DHGET. (X' and Y' are also honest because one can prove if $S[X, Y, salt] \neq \perp$ then $E[X] \neq \perp$ and $E[Y] \neq \perp$.) Moreover, the probability of the adversary queries this collision is less than or equal to the probability of the collision itself. To bound the collision probability, let X_i be the i-th DH share returned by DHGEN for $i \in [Q]$ where Q is the number of queries to DHGEN. The probability of collision is bounded as follows:

$$
\begin{aligned}
&\Pr\left[\exists i_1, i_2, i_3, i_4 \text{ s.t. } X_{i_1}^{E[X_{i_2}]} = X_{i_3}^{E[X_{i_4}]} \wedge \{i_1, i_2\} \neq \{i_3, i_4\}\right] \\
&= \Pr\left[\bigvee_{\substack{i_1,i_2,i_3,i_4 \\ \{i_1,i_2\}\neq\{i_3,i_4\}}} X_{i_1}^{E[X_{i_2}]} = X_{i_3}^{E[X_{i_4}]}\right] \\
&\leq \sum_{\substack{i_1,i_2,i_3,i_4 \\ \{i_1,i_2\}\neq\{i_3,i_4\}}} \Pr\left[X_{i_1}^{E[X_{i_2}]} = X_{i_3}^{E[X_{i_4}]}\right] \\
&= \sum_{\substack{i_1,i_2,i_3,i_4 \\ \{i_1,i_2\}\neq\{i_3,i_4\}}} \Pr\left[g^{E[X_{i_1}]E[X_{i_2}]} = g^{E[X_{i_3}]E[X_{i_4}]}\right] \\
&= \sum_{\substack{i_1,i_2,i_3,i_4 \\ \{i_1,i_2\}\neq\{i_3,i_4\}}} \Pr\left[E[X_{i_1}]E[X_{i_2}] = E[X_{i_3}]E[X_{i_4}] \mod q\right]
\end{aligned}
$$

where the last equality follows from the fact that order of g is q. Now, notice for each quadruple (i_1, i_2, i_3, i_4), $|\{i_1, i_2, i_3, i_4\}| \geq 2$. For each of these quadruples, one can assume without loss of generality that $i_1 \neq i_3$. Hence,

$$
\begin{aligned}
&\Pr_{E[X_{i_1}],E[X_{i_3}],\ldots}\left[E[X_{i_1}]E[X_{i_2}] = E[X_{i_3}]E[X_{i_4}] \mod q\right] \\
&\quad = \Pr_{E[X_{i_1}],E[X_{i_3}],\ldots}\left[E[X_{i_1}] = E[X_{i_3}]E[X_{i_4}]E[X_{i_2}]^{-1} \mod q\right] \\
&\quad = \sum_e \Pr_{E[X_{i_1}],E[X_{i_3}],\ldots}\left[E[X_{i_1}] = E[X_{i_3}]E[X_{i_4}]E[X_{i_2}]^{-1} \mod q | E[X_{i_3}] = e\right] \Pr_{E[X_{i_3}]}\left[E[X_{i_3}] = e\right] \\
&\quad = \frac{1}{q}\sum_e \Pr_{E[X_{i_1}],E[X_{i_3}],\ldots}\left[E[X_{i_1}] = E[X_{i_3}]E[X_{i_4}]E[X_{i_2}]^{-1}] \mod q | E[X_{i_3}] = e\right] \\
&\quad \leq \frac{1}{q}\sum_e \frac{1}{q} \\
&\quad = \frac{1}{q}
\end{aligned}
$$

Notice that $\Pr_{E[X_{i_1}],E[X_{i_3}],\ldots}\left[E[X_{i_1}] = E[X_{i_3}]E[X_{i_4}]E[X_{i_2}]^{-1} \mod q | E[X_{i_3}] = e\right]$ is $\frac{1}{q}$ when $|\{i_1, i_2, i_3, i_4\}| = 2$ (because $\{i_2, i_4\} = \{i_1, i_3\}$), $\frac{1}{q^2}$ when $|\{i_1, i_2, i_3, i_4\}| = 3$,

and $\frac{1}{q^4}$ when $|\{i_1, i_2, i_3, i_4\}| = 4$. Therefore, the collision probability is bounded by $\frac{Q^4-2Q^2+Q}{q}$ where $Q^4 - 2Q^2 + Q$ is the number of quadruples where $\{i_1, i_2\} \neq \{i_3, i_4\}$. (There are Q^4 in total, Q of which are of the form (a, a, a, a) while $4\binom{Q}{2}$ ones are either of the forms (a, b, a, b) or (a, b, b, a).)

To get a tighter bound, notice that there are exactly $6\binom{Q}{2}$ quadruples where $\{i_1, i_2\} \neq \{i_3, i_4\}$ and $|\{i_1, i_2, i_3, i_4\}| = 2$. ($4\binom{Q}{2}$ quadruples for the case of $\{a, b\} \neq \{a, a\}$ and $2\binom{Q}{2}$ quadruples for the case of $\{a, a\} \neq \{b, b\}$). There are $3\times3!\times4\times\binom{Q}{3}/2$ quadruples where $|\{i_1, i_2, i_3, i_4\}| = 3$. Finally, $4! \times \binom{Q}{4}$ quaruples where $|\{i_1, i_2, i_3, i_4\}| = 4$. Hence the collision probability can be bounded by $\frac{6\binom{Q}{2}}{q} + \frac{36\binom{Q}{3}}{q^2} + \frac{24\binom{Q}{4}}{q^3}$. □

5.2.2 Proof of Lemma 5.3

Proof of Lemma 5.3. The advantage of adversary is bounded by the probability of assertion failure which is $\frac{1}{q}$. Notice that in a prime-ordered group G, where $|G| = q$, for $1 \leq \alpha < q$, we have $S^\alpha = T^\alpha$ if and only if $S = T$. (If $S^\alpha = T^\alpha$, then $(ST^{-1})^\alpha = \mathsf{id}(G)$. Assume $ST^{-1} \neq \mathsf{id}(G)$, by applying Lagrange theorem to the order of cyclic group generated by ST^{-1}, we conclude $\alpha \mid q$ which implies $\alpha = 1, q$.) □

5.2.3 Proof of Lemma 5.4

Proof of Lemma 5.4. This is an example of statistical indistinguishability because even unbounded adversaries can not distinguish the real and ideal games. Let $R = (g^{X_1}, \ldots, g^{X_Q}, X_1, \ldots, X_Q)$ denotes the information the adversary gets after interacting with game for Q queries to DHGEN in the real game and similarly $I = (g^{AX_1}, \ldots, g^{AX_Q}, AX_1, \ldots, AX_Q)$ be the information the adversary gets from the ideal game where X_i's and A are random variables and g^{X_i}'s (g^{AX_i}'s) be the outputs of DHGEN and X_i's (AX_i's) be the outputs of DHGET. We can show for any $(y_1, \ldots, y_Q) \in Z_q^Q$,

$$\Pr\left[R = (g^{y_1}, \ldots, g^{y_Q}, y_1, \ldots, y_Q)\right] = \Pr\left[I = (g^{y_1}, \ldots, g^{y_Q}, y_1, \ldots, y_Q)\right]$$

and conclude the distribution of information the adversary receives in the real and ideal games are exactly the same. Clearly, $\Pr\left[R = (g^{y_1}, \ldots, g^{y_Q}, y_1, \ldots, y_Q)\right] = \frac{1}{q^Q}$.

On the other hand,

$$\begin{aligned}
&\Pr\left[I = (g^{y_1}, \ldots, g^{y_Q}, y_1, \ldots, y_Q)\right] \\
&\quad = \sum_{\alpha} \Pr_{A,X_i}\left[I = (g^{y_1}, \ldots, g^{y_Q}, y_1, \ldots, y_Q) | A = \alpha\right] \Pr[A = \alpha] \\
&\quad = \sum_{\alpha} \Pr_{A,X_i}\left[g^{AX_1} = g^{y_1} \wedge \cdots \wedge g^{AX_Q} = g^{y_Q} \wedge AX_1 = y_1 \wedge \cdots \wedge AX_Q = y_Q | A = \alpha\right] \Pr[A = \alpha] \\
&\quad = \frac{1}{q} \sum_{\alpha} \Pr_{A,X_i}\left[g^{AX_1} = g^{y_1} \wedge \cdots \wedge g^{AX_Q} = g^{y_Q} \wedge X_1 = A^{-1}y_1 \wedge \cdots \wedge X_Q = A^{-1}y_Q | A = \alpha\right] \\
&\quad = \frac{1}{q} \sum_{\alpha} \frac{1}{q^Q} \\
&\quad = \frac{1}{q^Q}
\end{aligned}$$

Hence, for uniformly chosen α and x_i's, we have

$$(g^{\alpha x_1}, \ldots, g^{\alpha x_Q}, \alpha x_1, \ldots, \alpha x_Q) \overset{stat}{\approx} (g^{x_1}, \ldots, g^{x_Q}, x_1, \ldots, x_Q).$$

□

Remark. For formal verification of the entire proof in SSBee, one shall define indistinguishability of games $Ghcoll^{b,grp}$, $Gccoll^{b,grp}$, $Gstat^{b,grp}$, $Galpha^{b,grp}$ as asssumptions because only the reductions of the hybrid games G_i to these games can be verified in SSBee. On the other hand, reductions $G_5 \overset{code}{\equiv} \mathcal{R}_{56} \to G'_5$ and $G_6 \overset{code}{\equiv} \mathcal{R}_{56} \to G'_6$ as well as all other code equivalences in Claims 5.8 and 5.19 can be essentially verified in SSBee. (with a proper encoding of the loops)

References

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. *The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES*, page 143–158. Springer Berlin Heidelberg, 2001.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. Drown: breaking tls using sslv2. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 689–706, USA, 2016. USENIX Association.

[AZ24] Jeremy Avigad and Richard Zach. The Epsilon Calculus. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2024 edition, 2024.

[BBB+19] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. Cryptology ePrint Archive, Paper 2019/1393, 2019.

[BBB+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[BBDL+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, 2015.

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BCDS22] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: Verification of Large-Scale, Real World, Cryptographic Protocols. *IEEE Security and Privacy Magazine*, 2022.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the MLS RFC, draft 11. Cryptology ePrint Archive, Paper 2021/137, 2021.

[BDLE+21] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Paper 2021/467, 2021. https://eprint.iacr.org/2021/467.

[BDLF+18] Chris Brzuska, Antoine Delignat-Lavaud, Cedric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. Cryptology ePrint Archive, Paper 2018/306, 2018. https://eprint.iacr.org/2018/306.

[BEW25] Chris Brzuska, Christoph Egger, and Jan Winkelmann. Ssbee, 2025. https://github.com/sspverif/sspverif.

[BFGJ17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. Cryptology ePrint Archive, Paper 2017/517, 2017. https://eprint.iacr.org/2017/517.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, page 71–90, Berlin, Heidelberg, 2011. Springer-Verlag.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009.

[BJ23] Bruno Blanchet and Charlie Jacomme. CryptoVerif: a Computationally-Sound Security Protocol Verifier. Technical Report RR-9526, Inria, October 2023.

[BL12] Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: the power of free precomputation. Cryptology ePrint Archive, Paper 2012/318, 2012.

[BL24] Chris Brzuska and Valtteri Lipiäinen. Companion to cryptographic primitives, protocols and proofs. https://cryptocompanion.github.io/cryptocompanion/cryptocompanion.pdf, 2024. Accessed: 2018-12-06.

[Bla12] Bruno Blanchet. Security protocol verification: symbolic and computational models. In *Proceedings of the First International Conference on Principles of Security and Trust*, POST'12, page 3–29, Berlin, Heidelberg, 2012. Springer-Verlag.

[Bla16] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1–2):1–135, October 2016.

[Bla18] Bruno Blanchet. Composition theorems for cryptoverif and application to tls 1.3. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 16–30, 2018.

[BO21] Chris Brzuska and Sabine Oechsner. A state-separating proof for yao's garbling scheme. Cryptology ePrint Archive, Paper 2021/1453, 2021.

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Paper 2004/331, 2004.

[BWM99] Simon Blake-Wilson and Alfred Menezes. Unknown key-share attacks on the station-to-station (sts) protocol. In *Public Key Cryptography, Second International Workshop on Practice and Theory in Public Key Cryptography, PKC '99, Kamakura, Japan, March 1-3, 1999, Proceedings*, volume 1560 of *Lecture Notes in Computer Science*, pages 154–170. Springer, 1999.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Paper 2000/067, 2000.

[CCD+23] Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. Hash gone bad: Automated discovery of protocol attacks that exploit hash function weaknesses. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5899–5916, Anaheim, CA, August 2023. USENIX Association.

[CDJZ23] Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. Automated analysis of protocols that use authenticated encryption: How subtle AEAD differences can impact protocol security. Cryptology ePrint Archive, Paper 2023/1246, 2023.

[CHH+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.

[CHSvdM16] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of tls 1.3: 0-rtt, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485, 2016.

[CJ19] Cas Cremers and Dennis Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman. Cryptology ePrint Archive, Paper 2019/526, 2019.

[CS98] Ronald Cramer and Victor Shoup. *A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack*, page 13–25. Springer Berlin Heidelberg, 1998.

[CVE09] Cve-2009-3555., 2009. https://www.cve.org/CVERecord?id=CVE-2009-3555.

[DFGS20] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. Cryptology ePrint Archive, Paper 2020/1044, 2020.

[DG19] Nir Drucker and Shay Gueron. Selfie: reflections on TLS 1.3 with PSK. Cryptology ePrint Archive, Paper 2019/347, 2019.

[DKO21] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. Bringing state-separating proofs to EasyCrypt - a security proof for cryptobox. Cryptology ePrint Archive, Paper 2021/326, 2021.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Egg22] Christoph Egger. *On Abstraction and Modularization in Protocol Analysis*. Phd thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2022. https://nbn-resolving.org/urn:nbn:de:bvb:29-opus4-238950.

[EMS25] Ross Evans, Matthew McKague, and Douglas Stebila. ProofFrog: A tool for verifying game-hopping proofs. Cryptology ePrint Archive, Paper 2025/418, 2025.

[Eva24] Ross Evans. Prooffrog: A tool for verifying game-hopping proofs. Master thesis, University of Waterloo, 2024. http://hdl.handle.net/10012/20441.

[FG17] Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 60–75, 2017.

[HRM+21] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in coq. Cryptology ePrint Archive, Paper 2021/397, 2021.

[JCCGS19] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. Cryptology ePrint Archive, Paper 2019/779, 2019.

[JKSS11] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. Cryptology ePrint Archive, Paper 2011/219, 2011.

[KE10] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[Koh23] Konrad Kohbrok. *State-Separating Proofs and Their Applications*. Phd thesis, Aalto University, 2023. https://urn.fi/URN:ISBN:978-952-64-1356-3.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. Cryptology ePrint Archive, Paper 2010/264, 2010.

[KW15] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. Cryptology ePrint Archive, Paper 2015/978, 2015.

[Lei10] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[Mau11] Ueli Maurer. Constructive cryptography — a new paradigm for security definitions and proofs. In *Proceedings of the 2011 International Conference on Theory of Security and Applications*, TOSCA'11, page 33–56, Berlin, Heidelberg, 2011. Springer-Verlag.

[MR11] Ueli Maurer and Renato Renner. Abstract cryptography. In Bernard Chazelle, editor, *The Second Symposium on Innovations in Computer Science, ICS 2011*, pages 1–21. Tsinghua University Press, 1 2011.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[Pun21] Kirthivaasan Puniamurthy. A proof viewer for state-separating proofs: Yao's garbling scheme. Master thesis, Aalto University, 2021. https://urn.fi/URN:NBN:fi:aalto-202101311851.

[Raj25a] Amirhosein Rajabi. Kem-dem cca security formalization in ssbee, 2025. https://github.com/tornado80/sspverif/tree/kem-dem-cca-security/example-projects/kem-dem-cca-security.

[Raj25b] Amirhosein Rajabi. Parameterless tls 1.3 key schedule security formalization in ssbee (second approach), 2025. https://github.com/tornado80/sspverif/tree/tls13-key-schedule/example-projects/tls13-key-schedule-parameterless.

[Raj25c] Amirhosein Rajabi. Parametrized tls 1.3 key schedule security formalization in ssbee (first approach), 2025. https://github.com/tornado

80/sspverif/tree/tls13-key-schedule/example-projects/tls13-key-schedule.

[Raj25d] Amirhosein Rajabi. Sodh g5g6 equivalence, 2025. https://github.com/tornado80/sspverif/tree/sodh-stuff/example-projects/sodh-g5g6-equivalence.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[Roc25] Rocq Prover Development Team. The Rocq prover reference manual – release 9.0.0. https://rocq-prover.org/doc/V9.0.0/refman/index.html, 2025.

[Rog06] Phillip Rogaway. Formalizing human ignorance: Collision-resistant hashing without the keys. Cryptology ePrint Archive, Paper 2006/281, 2006.