Security in the Presence of Key Reuse: Context-Separable Interfaces and their Applications

Christopher Patton and Thomas Shrimpton

Florida Institute for Cybersecurity Research
Computer and Information Science and Engineering
University of Florida
{cjpatton,teshrim}@ufl.edu

Abstract. Key separation is often difficult to enforce in practice. While key reuse can be catastrophic for security, we know of a number of cryptographic schemes for which it is provably safe. But existing formal models, such as the notions of joint security (Haber-Pinkas, CCS '01) and agility (Acar et al., EUROCRYPT '10), do not address the full range of key-reuse attacks—in particular, those that break the abstraction of the scheme, or exploit protocol interactions at a higher level of abstraction. This work attends to these vectors by focusing on two key elements: the *game* that codifies the scheme under attack, as well as its intended adversarial model; and the underlying *interface* that exposes secret key operations for use by the game. Our main security experiment considers the implications of using an interface (in practice, the API of a software library or a hardware platform such as TPM) to realize the scheme specified by the game when the interface is shared with other unspecified, insecure, or even malicious applications. After building up a definitional framework, we apply it to the analysis of two real-world schemes: the EdDSA signature algorithm and the Noise protocol framework. Both provide some degree of *context separability*, a design pattern for interfaces and their applications that aids in the deployment of secure protocols.

Keywords: Key reuse, APIs, Diffie-Hellman, EdDSA, Noise

1 Introduction

The principle of *key separation*, or ensuring that distinct cryptographic functionalities use distinct keys, is a widely accepted tenet of applied cryptography. It appears to be difficult to follow, however, as there are many instances of *key reuse* in deployed cryptosystems, some having significant impact on the security of applications. There are a number of practical matters that lead to key resuse. First, operational requirements of the system often demand some degree of it. For example, it is common to use a signing key deployed for TLS [32] in other protocols, as this is permitted by certificate authorities and avoids the cost of certifying a distinct key for each protocol. But doing so has side effects that must be addressed in the design of these protocols, as well as the interface that exposes the key to applications [9]. Second, it is often not clear what constitutes a "distinct functionality". Intel's Trusted Platform Module (TPM) standard [36] supports a variety of protocols for remote attestation that use an Intel-certified key stored on chip. The TPM exposes a core set of operations involving this key via its application-programming interface (API), which applications make calls to in order to implement attestation schemes. But the requirement to support so many protocols has lead to a flexibile API with subtle vulnerabilities [2,12].

Prior work sheds light on when key reuse is safe among specific primitives. Haber and Pinkas [16] introduce the notion of *joint* security, which captures the security of a target cryptosystem (say, a digital signature scheme) in the presence of an oracle that exposes a related secret-key operation (say, the decryption operation of a public-key encryption scheme). Many widely used primitives are jointly secure, including RSA-PSS/OAEP [16] and Schnorr signatures/hybrid encryption [13]. Acar et al. [1] address the related problem of *agility*, where the goal is to identify multiple instantiations of a particular primitive (e.g., sets of AEAD schemes, PRFs, or signature schemes) that can securely use the same key material. But the range of potential key-reuse attacks goes well beyond what these works cover; attack vectors sometimes break the intended abstraction boundary of the scheme by exposing lower level operations [11,2], or involve unforeseen protocol interactions at a higher level of abstraction [18,9]. We believe that a comprehensive treatment of key reuse can and should account for these attack vectors as well.

To this end, we propose to surface the API as a first class security object. For our purposes, the API (or just "interface") is the component of a system that exposes to applications a fixed set of operations involving one or more secret keys. APIs are often the root-of-trust of applications: TPM, Intel's Software Guard Extensions (SGX), hardware security modules (HSMs), and even chip-and-pin credit cards all provide cryptographic APIs that aim to be trustworthy-by-design. But pressure to meet operational requirements, while exporting interfaces that are suitable for a variety of applications, often leads to vulnerabilities [10,13,21,2]. An analogous situation arises in the development of software that uses a cryptographic library; software engineers tend to trust that any use case *permitted* by an API is secure, without fully grasping its side-effects [27]. This phenomenon tends to lead to vulnerable code [3,28].

In light of these issues, this work seeks to develop security-oriented design principles for interfaces and their applications. We devise a definitional framework for reasoning about the security of an application when the interface it consumes is used in other, perhaps unintended or even insecure ways. We model these "other applications" very conservatively, as follows: to assist it in its attack against the target application, we assume the adversary has *direct* access to the underlying interface, allowing it to mount *exposed interface attacks* on a target application. We apply this framework to the design and analysis of two real-world cryptosystems: the EdDSA signature algorithm [17] and the Noise protocol framework [30]. In doing so, we elicit a property of interfaces and their applications we call *context separability*, which we will show to be an invaluable tool for secure protocol design.

The full version [29]. This is an extended abstract; the full version of this paper includes all deferred proofs, as well as additional results, remarks, and discussion.

The framework. We begin by motivating our definitional viewpoint, which draws abstraction boundaries a bit differently than usual. Game-based notions of security [6] typically specify (in pseudocode) a game $\mathcal{G}$ that makes calls to a cryptographic scheme Π (a primitive or protocol, also specified in pseudocode). The game captures an attack model—that is, the capabilities and goal of the adversary—and establishes boundaries on the permitted uses of Π. Model-specific adversarial capabilities are captured as oracle procedures specified by $\mathcal{G}$, which the adversary may query during its attack. Its goal is formalized by an explicit winning condition that depends on its queries and the random choices of the game. The security of the scheme, when used as specified by $\mathcal{G}$, is measured by executing an adversary with $\mathcal{G}$.

Suppose that Π is specified in terms of calls to an underlying interface $\mathcal{I}$, which defines the set of operations that can be performed on the secret key. Our goal is to measure the security of Π in the sense of $\mathcal{G}$ when the adversary playing the game is also provided direct access to $\mathcal{I}$, i.e., when the adversary is able to mount exposed interface attacks on the security of Π that $\mathcal{G}$ codifies.

We formalize our syntax for interfaces and games in Section 3. Rather than refer explicitly to Π, we allow the game $\mathcal{G}$ to realize Π as pseudocode that makes calls to $\mathcal{I}$. Interfaces may expose conventional primitive operations like signing or decryption, or they may expose lower level operations that are composed into higher level ones by the game. (This is precisely what TPM does; more on this in Section 5.1.) Our syntax for interfaces admits operations on symmetric and asymmetric keys. In the latter case, all secret-key operations are handled by the interface, and all public-key operations are specified by the game.

Security under exposed interface attack. The objects of our study are an interface and a target application; we formalize the latter as a game that defines the scheme, how it is used, and what is its goal. With some details suppressed, Figure 1 visualizes the execution flow of our main security experiment SEC/I, which acts as an analysis harness for an interface $\mathcal{I}$, game $\mathcal{G}$, and adversary $\mathcal{A}$. The experiment first generates the public and secret keys (pk, sk) as specified by $\mathcal{I}$, then runs $\mathcal{A}$ on input of pk and with access to oracles **Init**, **Call**, and **Final** used to "play" the game $\mathcal{G}$. The game is comprised of three algorithms: the first, $\mathcal{G}$.Init, takes pk as input and outputs the game's initial state; the second, $\mathcal{G}$.Call, specifies the capabilities of $\mathcal{A}$ in the game and advances the state in response to its queries; and the last, $\mathcal{G}$.Final, computes the game's winning condition and outputs a bit *win*. Both $\mathcal{G}$.Call and $\mathcal{G}$.Final are given access to $\mathcal{I}$ for performing secret key operations, and the adversary is given direct access to $\mathcal{I}$ via a fourth oracle **Op**. As usual [6], the adversary must call **Init** first and **Final** last; the outcome of the experiment is the value of *win*.

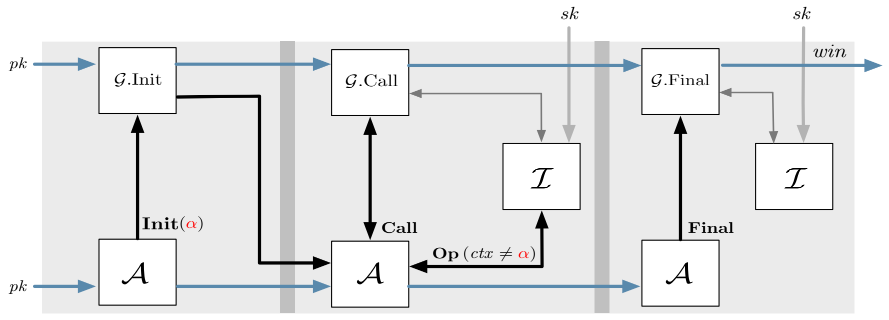

Fig. 1. Illustration of the SEC/I experiment, which has three "phases": first, the adversary $\mathcal{A}$ chooses the game context α and initializes the game $\mathcal{G}$; second, $\mathcal{A}$ plays $\mathcal{G}$ and interacts with $\mathcal{I}$; and third, $\mathcal{A}$ finalizes $\mathcal{G}$ and the experiment outputs the outcome *win*.

The central goal of our work is to measure the security "gap" between this and the "usual setting" in which the underlying interface is only used for the target application. This setting is formalized by the SEC experiment, which is defined just like SEC/I, except the adversary is denied access to **Op**. We will formalize both experiments in Section 4.

Context separability. Security in our setting often requires a property we call context separability. Loosely, a context-separable interface is one whose operations can be bound to the context in which they are used. When *context separation* is enforced, this binding prevents context-separable *games* from interacting in unintended ways. Let us consider an illustrative example. TLS is designed to prevent signatures produced in the context of the protocol from being used in other applications, and vice versa. To accomplish this, whenever a message is to be signed, it is signed together with a short *context string* that uniquely identifies the protocol version and the signer (i.e., the client or server, see [32, Section 4.4.3]). This makes it unlikely that another protocol would *inadvertently* produce a signature that could be used in TLS, but nothing about the protocol or the signature scheme ensures this; depending on how signing operations are exposed and whether key separation is enforced, this could lead to practical cross-protocol attacks [9].

As reflected in both our syntax and security notions, our framework sheds formal light on the affect of these design challenges on security. In addition to the secret key and operand, an interface is formalized to take as input a context string ctx, which is meant to uniquely identify the application making the API call; correspondingly, a game is initialized with context that is meant to uniquely identify it. In the SEC/I experiment, the game $\mathcal{G}$ is initialized with an adversarially chosen *game context* string α, which the adversary may not use for its interface queries. (See Figure 1.) This is akin to enforcing non-repeating nonces in the security experiment for symmetric encryption; in practice, it is an operational requirement that the environment must enforce.

On the role of context separation. The high-level goal of our work is to provide a framework for reasoning about the security of interfaces that expose secrets to applications. We uncover context separability as a useful design pattern for achieving security in the presence of key reuse. In fact, this operational requirement can be seen as a generalization of key separation; an interface could enforce key separation by generating a unique key for each unique application (identified by a context string) it intends to support. But when doing so is infeasible, interfaces and their applications can be designed so that reuse is secure as long as context separation is enforced.

We stress that context separation is not essential to security in the presence of key reuse. We could have formalized other operational requirements; it may suffice to ensure that no single operation is used in multiple applications, or that distinct applications provide distinct inputs, etc. However, our choice to enforce context separation in the SEC/I experiment was not arbitrary. First and foremost, it reflects a design pattern often explicit (but sometimes implicit) in real standards, two of which we analyze in this paper (EdDSA and Noise). Second, it is our hope that clarifying this simple requirement will reduce some of the complexity inherent to protocol design.

A composition theorem. To measure the "gap" between SEC and SEC/I—that is, to measure the security impact of exposing the underlying interface—in Section 4.2 we formulate and prove sufficiency of a condition under which security in the former sense implies security in the latter. The GAP1 experiment is associated to an interface $\mathcal{I}$, a game $\mathcal{G}$, a simulator $\mathcal{S}$, and a distinguisher $\mathcal{D}$. The experiment allows $\mathcal{D}$ to play the game via **Init**, **Call**, and **Final** as above; likewise, the adversary can query the interface via **Op**. In the "real" world, **Op** exposes $\mathcal{I}$, but in the "simulated" world, the distinguisher's queries are evaluated by $\mathcal{S}$, which is given the public key but *no* access to $\mathcal{I}$. The adversary's goal is to distinguish between these two worlds. We show that for any $\mathcal{I}$ and $\mathcal{G}$, if $\mathcal{I}$ is both SEC and GAP1 secure for $\mathcal{G}$, then $\mathcal{I}$ is also SEC/I secure for $\mathcal{G}$ (Theorem 1(i)). Thus, proving GAP1 security of $\mathcal{I}$ for $\mathcal{G}$ will be our primary goal, as it succinctly characterizes conditions under which it is safe to compose applications that share the same interface.

We also consider the security impact of changing an interface, by, for example, exposing additional operations on the key. The GAP2 experiment is similar to GAP1, except it involves a *pair* of interfaces $(\mathcal{I}^1, \mathcal{I}^0)$. In the "real" world, both the game and distinguisher are given oracle access to $\mathcal{I}^1$; in the "simulated" world, the game is given an oracle for $\mathcal{I}^0$ and the distinguisher's **Op** queries are answered by the simulator, which is also given an oracle for $\mathcal{I}^0$. We prove that if $(\mathcal{I}^1, \mathcal{I}^0)$ is GAP2 secure for $\mathcal{G}$ and $\mathcal{I}^0$ is SEC/I secure for $\mathcal{G}$, then so is $\mathcal{I}^1$ (Theorem 1(ii)). We also formulate a *necessary* condition, wGAP2, that allows us to characterize key operations that are not generally safe to expose in an interface.

Application to discrete log interfaces. We apply our framework to various *discrete log (DL) interfaces*, whose key pairs are $(p = g^s, s)$ where g is the generator of a finite, cyclic group. They are so named because the security of their applications is predicated on the hardness of computing discrete logarithms (in particular, $s = \log_g p$) in the given group. They are particularly interesting in our setting because they admit a wide variety of primitives and protocols.

Diffie-Hellman and EdDSA. A well-known design challenge for DL interfaces is avoiding accidental exposure of a static Diffie-Hellman (DH) oracle [2,12]: given p and an oracle that on input of q returns q^s, there is an algorithm [11] for computing s that is much faster than generic DL [31]. As a first exercise of our framework, we rule out the security of (inadvertently) exposing static DH in *any* DL interface by proving wGAP2 insecurity of their composition (Section 5.1). We then consider the security of the EdDSA signature scheme [8] in our setting (Section 5.2). The standardized version of this algorithm [17] admits variants that are context separable, allowing us to prove in the random oracle model (ROM) [5] that the signing operation is GAP1 secure for any game in which all signing and verification operations use the game context. We also show (in the ROM) that exposing the signing operation of *any* EdDSA variant in a DL interface that meets certain requirements is GAP2 secure in general.

Noise. Having addressed the security of these relatively simple operations, in Section 6 we turn to analyzing Noise [30], a framework for designing two-party secure-channel protocols. Participants in these protocols negotiate and execute *handshake patterns*, which define the sequence of messages sent between them and thereby the security of the communication channel they establish. We specify as an interface the set of *processing rules* that determine how each party consumes and produces messages, and how their state is updated as a side-effect. This allows handshake patterns to be executed by making calls to this interface.

Our results for Noise are largely positive. With a simple tweak of the processing rules, we are able to prove GAP1 security of our interface while making only minimal (and natural) assumptions about the target application. This implies, in particular, that all handshake patterns that can be executed by our interface are jointly secure (up to context separation). We cannot support all patterns, however, because some give rise to GAP1 distinguishing attacks in any interface that could be used to implement them. As a result of these limitations, our analysis leaves the security of key-reuse in Noise *as it is* an open question. Nevertheless, our work shows that Noise's approach to protocol design makes it possible to reason about protocol interactions in a very general way.

Finally, in the full version of this paper [29], we will directly address the composition of the security of using a key deployed for EdDSA in Noise (and vice versa).

Limitations of the framework. Our syntax for games is such that a wide variety of security goals can be expressed with them. However, the execution semantics of games in the SEC/I experiment excludes some important settings, including the multi-user setting [7] and those captured by multi-stage adversaries [33]. In the full version of this paper [29] we will briefly discuss how to formalize these settings as extensions to the SEC/I experiment. In addition, our interfaces are all *stateless*, which we found necessary for composition in general. (This is in line with prior works that address related problems [33].)

Related work. Here we highlight the works that inspired or are technically related to our framework and leave a broader overview of this area to the full version [29]. Our framework generalizes the setting of Shrimpton, Stam, and Warinschi [35], who study HSMs implementing the PKCS#11 standard for cryptographic APIs [15]. Their formulation of a "primitive" is closely related to our formulation of interfaces, and their framework allows for expressing arbitrary security goals for primitives, as ours does for interfaces.

Our security goals are reminiscent of joint security, and many of the proof techniques we use are borrowed from that area [16,13]. However, our notions are ultimately incompatible with theirs. To adapt our framework to the consideration of joint security, one would partition the set of operations exposed by the interface into those available to the target system (i.e., the game) and those available to the adversary.

The GAP2 notion can be viewed as a restricted form of indifferentiability [24]. In particular, the GAP2 experiment for $(\mathcal{I}^1, \mathcal{I}^0)$, $\mathcal{G}$, adversary $\mathcal{A}$, and simulator $\mathcal{S}$ is equivalent to the indifferentiability of $(\mathcal{I}^1, \mathcal{I}^0)$ with respect to the *specific* distinguisher $\mathcal{D}$ that is the composition of $\mathcal{G}$ and $\mathcal{A}$ prescribed by the GAP2 experiment. To be clear, this does *not* allow us to directly use the indifferentiability composition theorem. Our own result is about composing game $\mathcal{G}$ with interfaces $\mathcal{I}^1$ and, separately, $\mathcal{I}^0$; and although our composition theorem looks quite similar to [33, Theorem 1], the things being composed are not the same.

2 Pseudocode and Conventions

This section enumerates our conventions for pseudocode, algorithms, adversaries, and experiments. The reader may wish to skip this section and refer to it later as needed.

Pseudocode. Our pseudocode is based on Rogaway and Stegers [34]. Variables are statically typed. Available types are **set** (a set), **tup** (a tuple), **bool** (an element of $\{0,1\}$), **int** (an element of $\mathbb{Z}$), and **str** (an element of $\{0,1\}^*$). In general, if $X \in \mathcal{X}$, then we say that X has type $\mathbf{elem}_{\mathcal{X}}$. Variables are declared with the keyword **dec**, e.g., **dec int** x; **str** A. Variables need not be explicitly declared, in which case their type must be inferable from their initialization (i.e., the first use of the variable in an assignment statement). There are two *compound* types. The first is associative arrays, denoted by "[ ]", which map tuples (that is, a finite sequence of quantities of any type) to values of a specific type. For example, **dec str** $\pi[\,]$ declares an associative array π whose values are strings. We let $\pi[k]$ and π_k denote the value in π associated with k. The second is **struct**, which is used to recursively define new types; see Figure 7 for an example. We will also refer to the type of a procedure (i.e., an algorithm) by its interface. For instance, the type $\mathcal{A}(\mathbf{str}\, X, Y) \mapsto (\mathbf{int}\, i, \mathbf{str}\, A)$ indicates that $\mathcal{A}$ takes as input a pair of strings (X, Y) and outputs an integer i and a string A.

Nil and bottom. Uninitialized variables implicitly have the value $\diamond$, read "nil". If a variable of one type is set to a value of another type, then the variable takes the value $\diamond$. The symbol $\diamond$ is interpreted as $\emptyset$ in an expression involving sets, as the 0-length tuple in an expression involving tuples, as 0 (i.e., false) in a boolean expression, as 0 in an expression involving integers, and as ε in an expression involving strings. A non-**bool** variable X is interpreted as "$(X \neq \diamond)$" (i.e., "X is defined") in a boolean expression. If X is an associative array, then $X \leftarrow \diamond$ "resets" the array so that $X_k = \diamond$ for all k. Likewise, if X is a **struct**, then $X \leftarrow \diamond$ sets each field of X to $\diamond$. The symbol $\bot$, read "bottom", can be assigned to any variable regardless of type. Unlike $\diamond$, its interpretation in an expression is always undefined, except that $X = \bot$ and $\bot = X$ should evaluate to true just in case the previous assignment to X was $\bot$. (We remark that $\bot$ has the usual semantics in cryptographic pseudocode.)

Represented groups. We say that a group $\mathbb{G}$ is *represented* if $\diamond \notin \mathbb{G}$. We define an additional type, $\mathbf{elem}_{\mathbb{G}}$, parameterized by a represented group $\mathbb{G}$. We emphasize that, unlike **set**, **tup**, **bool**, **int**, or **str**, using the symbol $\diamond$ in an expression involving values of this type is not well-defined, since $\diamond$ has no interpretation as an element of $\mathbb{G}$.

Refined types. Variable declarations may be written as set-membership assertions. For example, **dec int** s; $\mathbf{elem}_{\mathbb{G}}\, P$ may be written like **dec** $s \in \mathbb{Z}$; $P \in \mathbb{G}$. Where appropriate, these types may also be refined, e.g. **dec** $s \in \mathbb{N}$.

String and tuple operations. Let $|X|$ denote the length of a string (or tuple) X. We denote the i-th element of X by X_i or $X[i]$. We define $X \,\|\, Y$ to be the concatenation of X with string (or tuple) Y. Let $X[i{:}j]$ denote the sub-string (or sub-tuple) $X_i \,\|\, \cdots \,\|\, X_j$ of X. If $i \notin [1..j]$ or $j \notin [i..|X|]$, then define $X[i{:}j] = \diamond$. Let $X[i{:}] = X[i{:}|X|]$ and $X[{:}j] = X[1{:}j]$.

Encoding of types. A value of any type can be encoded as a string. We will not define this encoding explicitly, but assume it possesses the following properties. Let $\underline{x_1, \ldots, x_m}$ denote the encoding of a tuple $(x_1, \ldots, x_m)$ as a string. Decoding is written as $\underline{x_1, \ldots, x_m} \leftarrow X$ and works like this (slightly deviating from [34, Section 2]): if there exist $y_1, \ldots y_n$ such that $X = \underline{y_1, \ldots, y_n}$, $m = n$, and each y_i has the same type as x_i, then set $x_i \leftarrow y_i$ for each $1 \leq i \leq m$. Otherwise, set $x_i \leftarrow \diamond$ for each $1 \leq i \leq m$. Let $\underline{x}_n$ denote the encoding of an integer $x \geq 0$ as an n-bit string. We write $\underline{x}_n \leftarrow X$ to denote decoding X as an n-bit, non-negative integer and assigning it to x. Finally, we say that a group $\mathbb{G}$ is *v-encoded* if it is represented and for all $X \in \mathbb{G}$ it holds that $|\underline{X}| = v$.

Passing variables by reference. It is customary in cryptographic pseudocode to pass all variables by *value*; we also permit variables to be passed by *reference.* (This idea is due to Rogaway and Stegers [34], but our semantics deviates from theirs.) Specifically, variables passed to procedures may be embellished with the symbol "&". If the variable appears on the left hand side of an assignment statement, then this immediately changes the value of the variable; when used in an expression, the variable is treated as its value. A procedure's interface makes explicit each input that is passed by reference. For example, in a procedure $\mathcal{A}(\&\mathbf{int}\, x, \mathbf{int}\, y) \mapsto \mathbf{int}\, z$, variable y is passed by value, while x is passed by reference. For example, after executing $x, y \leftarrow 0$; $z \twoheadleftarrow \mathcal{A}(\&x, y)$, the value of x may be non-0, but y is necessarily equal to 0.

Algorithms, experiments, and adversaries. Algorithms are randomized unless stated otherwise. An algorithm is t-time if for every choice of random coins, the algorithm halts in at most t time steps.[1] When an algorithm $\mathcal{A}$ is deterministic we write $y \leftarrow \mathcal{A}(x)$ to denote executing $\mathcal{A}$ on input of x and assigning its output to y; if $\mathcal{A}$ is randomized, then we write $y \twoheadleftarrow \mathcal{A}(x)$. Let $[\mathcal{A}(x)]$ denote the set of possible outputs of $\mathcal{A}$ when run on input x. Algorithms may have access to one or more oracles, written as superscripts, e.g., $y \twoheadleftarrow \mathcal{A}^{\mathcal{O},\cdots}(x)$. When this notation becomes cumbersome we may write $y \twoheadleftarrow \langle \mathcal{A} \colon \mathcal{O}, \ldots \rangle(x)$ instead. When we specify a procedure, if the procedure halts without an explicit **ret**-statement (i.e., a "return" statement), then it returns $\perp$.

We regard security experiments as algorithms whose output is always a bit. If "XXX" is an experiment associated with an adversary $\mathcal{A}$, we write $\mathbf{Exp}^{\mathrm{xxx}}(\mathcal{A})$ to denote the event that the experiment is run with $\mathcal{A}$ and the output is 1, i.e., $\Pr\left[\,\mathbf{Exp}^{\mathrm{xxx}}(\mathcal{A})\,\right]$ denotes the probability that XXX run with $\mathcal{A}$ outputs 1, where the probability is over the coins of XXX and $\mathcal{A}$. An adversary is an algorithm associated to a security experiment in which it is executed exactly once. (Thus, in this paper we restrict ourselves to the single-stage adversary setting [33].) Our convention will be that an adversary is t-time if its experiment is t-time. That is, an XXX-adversary $\mathcal{A}$ is t-time if $\mathbf{Exp}^{\mathrm{xxx}}(\mathcal{A})$ is t-time.

Miscellaneous. Logarithms are base-2 unless the base is given explicitly. If $\mathcal{X}$ is a set, then we write $x \twoheadleftarrow \mathcal{X}$ to denote sampling x randomly from $\mathcal{X}$ according to some distribution associated to $\mathcal{X}$; if $\mathcal{X}$ is finite and the distribution is unspecified, then it is uniform.

3 Interfaces and Games

In this section we define the syntax for *interfaces* and *games*, the fundamental components of our framework. A game captures an attack model (the capabilities and goals of an adversary) as well as an intended use of cryptographic operations that are provided (via black-box calls) by an interface. Typically, this use will be to realize some cryptographic scheme (i.e., primitive or protocol) that is under attack.

Definition 1 (Interfaces). An *interface* is a pair of algorithms $\mathcal{I} = (\mathrm{Gen}, \mathrm{Op})$ defined as follows:

- $\mathrm{Gen}(\,) \mapsto \mathbf{str}\, pk, sk$. The *key generator* outputs pair of key strings.
- $\mathrm{Op}(\mathbf{str}\, sk, ctx, op, in) \mapsto \mathbf{str}\, out$. The *key operator* exposes operations involving the key sk. It takes as input the context ctx, the operation identifier op, and the operand in, and it outputs the result out.

For compactness, we may denote $\mathcal{I}.\mathrm{Op}(sk, ctx, op, in)$ by $\mathcal{I}_{sk}(ctx, op, in)$ in the remainder. ⧫

[1] What constitutes a "time step" depends on the model of computation, which we leave implicit.

In our security experiments, the "public key" pk will be made available to all parties, but the "secret key" sk will be kept private by the interface. We note that $pk = \varepsilon$ is allowed, so that symmetric-key operations are within scope.

Definition 2 (Games). A *game* is a triple of algorithms $\mathcal{G} = (\mathrm{Init}, \mathrm{Call}, \mathrm{Final})$ defined as follows:

- $\mathrm{Init}(\mathbf{str}\ pk, \alpha) \mapsto \mathbf{str}\ st, out$. This is the game *initiator*. It takes as input the public key pk and game context α and outputs the initial state st and a string out.
- $\mathrm{Call}^{\mathcal{O}}(\&\mathbf{str}\ st, \mathbf{str}\ in) \mapsto \mathbf{str}\ out$. The *caller* is used to advance the state of an already initialized game. It abstracts all oracle queries except initialization and finalization. The first input is a reference to the game state, which may be updated as a side-effect of invoking the caller; the interpretation of the second input is up to the game. The caller expects access to an oracle $\mathcal{O}$, which we will call the *interface oracle*. It takes as input three strings and returns one.
- $\mathrm{Final}^{\mathcal{O}}(\mathbf{str}\ st, in) \mapsto \mathbf{bool}\ r$. The *finalizer* is used to decide if a game is in a winning state. Its inputs are the game state st and a string in, which is used to compute the winning condition. Oracle $\mathcal{O}$ is as defined for the caller.

For compactness, we occasionally denote $\mathcal{G}.\mathrm{Call}^{\mathcal{O}}(\&st, in)$ by $\mathcal{G}^{\mathcal{O}}_{st}(in)$. We say that $\mathcal{G}$ is *c-bound* if the caller and finalizer each make at most c calls to $\mathcal{O}$ during any one execution of the algorithm. ♦

4 Security Under Exposed Interface Attack

The goal of this work is to understand the security of cryptographic schemes when they are realized by an interface that may also be exposed to other, possibly insecure or (or even malicious) applications. The following experiment (SEC/I) captures this formally, allowing us prove or disprove security of a scheme (both codified by a game $\mathcal{G}$) when a given interface $\mathcal{I}$ is callable by both the game $\mathcal{G}$ and the adversary $\mathcal{A}$. An adversary in this experiment is said to be mounting an *exposed interface attack* on $\mathcal{G}$. We define another experiment (SEC) that captures the usual setting in which the adversary does not have this access.

Definition 3 (SEC/I and SEC security). Figure 2 defines two security experiments: SEC/I includes the boxed statement (but not the shaded one), and SEC includes the shaded statement (but not the boxed one). Both experiments begin by running the key generator $\mathcal{I}.\mathrm{Gen}$ and executing the adversary $\mathcal{A}$ on input of the public key and with access to four oracle procedures:

- **Init** initializes $\mathcal{G}$ by calling the initiator $\mathcal{G}.\mathrm{Init}$ on the public key and the game context chosen by $\mathcal{A}$ and returns the output out of the initiator.
- **Call** advances the game by invoking the caller $\mathcal{G}.\mathrm{Call}$ on input in provided by $\mathcal{A}$ and with oracle access to the interface $\mathcal{I}.\mathrm{Op}(sk, \cdot, \cdot, \cdot)$. It returns the output out of the caller.

```
$\boxed{\mathbf{Exp}^{\mathrm{sec/i}}_{\mathcal{I},\mathcal{G}}(\mathcal{A})}$ / $\mathbf{Exp}^{\mathrm{sec}}_{\mathcal{I},\mathcal{G}}(\mathcal{A})$
    dec str $sk, st, \alpha$; bool $win$
    $(pk, sk) \twoheadleftarrow \mathcal{I}.\mathrm{Gen}()$
    $\boxed{\langle \mathcal{A}\colon \mathbf{Init}, \mathbf{Final}, \mathbf{Call}, \mathbf{Op}\rangle(pk)}$
    $\langle \mathcal{A}\colon \mathbf{Init}, \mathbf{Final}, \mathbf{Call}\rangle(pk)$
    ret $win$

Init$(ctx)$
    $(st, out) \twoheadleftarrow \mathcal{G}.\mathrm{Init}(pk, ctx)$
    $\alpha \leftarrow ctx$; ret $out$

Final$(in)$
    $win \twoheadleftarrow \mathcal{G}.\mathrm{Final}^{\mathcal{I}.\mathrm{Op}(sk,\cdot,\cdot,\cdot)}(st, in)$
    ret $win$

Call$(in)$
    ret $\mathcal{G}.\mathrm{Call}^{\mathcal{I}.\mathrm{Op}(sk,\cdot,\cdot,\cdot)}(\&st, in)$

Op$(ctx, op, in)$
    if $ctx = \alpha$ then ret $\bot$
    ret $\mathcal{I}.\mathrm{Op}(sk, ctx, op, in)$
```

Fig. 2. The SEC/I and SEC experiments for interface $\mathcal{I}$, game $\mathcal{G}$, and adversary $\mathcal{A}$.

- **Op** exposes $\mathcal{I}.\mathrm{Op}(sk, \cdot, \cdot, \cdot)$ to $\mathcal{A}$ directly with the restriction that each query use a context string *ctx* that is different from the game context used to initialize the game.
- **Final** finalizes $\mathcal{G}$ by running the finalizer $\mathcal{G}$.Final on input *in* provided by $\mathcal{A}$ and setting *win* to the output and returning the value of *win* to $\mathcal{A}$.

The outcome of the experiment is the value of *win* when $\mathcal{A}$ halts. A valid SEC/I adversary makes a single query to **Init**, this being its first; it may then make any number of queries to **Call** and **Op**.[2] It completes its execution by making a single query to **Final**. We define the advantage of a (valid) SEC/I-adversary $\mathcal{A}$ in attacking $\mathcal{I}$ with respect to $\mathcal{G}$ as

$$\mathbf{Adv}^{\mathrm{sec/i}}_{\mathcal{I},\mathcal{G}}(\mathcal{A}) = \Pr\left[\,\mathbf{Exp}^{\mathrm{sec/i}}_{\mathcal{I},\mathcal{G}}(\mathcal{A})\,\right].$$

We call a SEC/I adversary (t, q_G, q_I)-resource if it is t-time and makes at most q_G and q_I queries to **Call** and **Op** respectively. We define the maximum advantage of any $\boldsymbol{r}$-resource SEC/I-adversary as $\mathbf{Adv}^{\mathrm{sec/i}}_{\mathcal{I},\mathcal{G}}(\boldsymbol{r})$. SEC security of $\mathcal{I}$ with respect to $\mathcal{G}$ is defined in kind, except that **Op** is not given to $\mathcal{A}$. We denote the advantage of SEC-adverseary $\mathcal{A}$ in attacking $\mathcal{I}$ with respect to $\mathcal{G}$ by $\mathbf{Adv}^{\mathrm{sec}}_{\mathcal{I},\mathcal{G}}(\mathcal{A}) = \Pr\left[\,\mathbf{Exp}^{\mathrm{sec}}_{\mathcal{I},\mathcal{G}}(\mathcal{A})\,\right]$, and we define $\mathbf{Adv}^{\mathrm{sec}}_{\mathcal{I},\mathcal{G}}(\boldsymbol{r})$ as above. Informally, we say that $\mathcal{I}$ is SEC/I (resp. SEC) secure for $\mathcal{G}$ if every efficient SEC/I (resp. SEC) adversary has small advantage.

Finally, if each of $\mathcal{G}$.Call's and $\mathcal{G}$.Final's interface queries is a triple (α, op, in) such that α is the context with which the game was initialized, then we say $\mathcal{G}$ is *regular* for SEC/I (resp. SEC). ⧫

Regular games and context separation. We remark that a game being regular is a property of the execution semantics of the game in the experiment, and

[2] Disallowing **Op** queries prior to **Init** is necessary for enforcing context separation. This restriction could be lifted by, say, allowing pre-**Init** access to **Op**, but demanding that none of these queries uses the (adversarially chosen) game context α.

```
$\mathbf{Exp}^{\mathrm{gap1}}_{\mathcal{I},\mathcal{G}}(\mathcal{S},\mathcal{D})$
  $\textbf{dec str}\ sk, st, \sigma, \alpha;\ b \twoheadleftarrow \{0,1\}$
  $(pk, sk) \twoheadleftarrow \mathcal{I}.\mathrm{Gen}(\,);\ \sigma \twoheadleftarrow \mathcal{S}.\mathrm{Init}(pk)$
  $d \twoheadleftarrow \langle \mathcal{D}\colon \mathbf{Init}, \mathbf{Final}, \mathbf{Call}, \mathbf{Op} \rangle(pk)$
  $\textbf{ret}\ (d = b)$

$\mathbf{Init}(ctx)$
  $(st, out) \twoheadleftarrow \mathcal{G}.\mathrm{Init}(pk, ctx)$
  $\alpha \leftarrow ctx;\ \textbf{ret}\ out$

$\mathbf{Final}(in)$
  $\textbf{ret}\ \mathcal{G}.\mathrm{Final}^{\mathcal{I}_{sk}}(st, in)$

$\mathbf{Call}(in)$
  $\textbf{ret}\ \mathcal{G}.\mathrm{Call}^{\mathcal{I}_{sk}}(\&st, in)$

$\mathbf{Op}(ctx, op, in)$
  $\textbf{if}\ ctx = \alpha\ \textbf{then ret}\ \bot$
  $\textbf{if}\ b = 1\ \textbf{then ret}\ \mathcal{I}_{sk}(ctx, op, in)$
  $\textbf{ret}\ \mathcal{S}.\mathrm{Op}^{\bot}(\&\sigma, ctx, op, in)$
```

```
$\mathbf{Exp}^{\mathrm{gap2}}_{\mathcal{I}^1,\mathcal{I}^0,\mathcal{G}}(\mathcal{S},\mathcal{D})$
  $\textbf{dec str}\ sk, st, \sigma, \alpha;\ b \twoheadleftarrow \{0,1\}$
  $(pk, sk) \twoheadleftarrow \mathcal{I}^b.\mathrm{Gen}(\,);\ \sigma \twoheadleftarrow \mathcal{S}.\mathrm{Init}(pk)$
  $d \twoheadleftarrow \langle \mathcal{D}\colon \mathbf{Init}, \mathbf{Final}, \mathbf{Call}, \mathbf{Op} \rangle(pk)$
  $\textbf{ret}\ (d = b)$

$\mathbf{Init}(ctx)$
  $(st, out) \twoheadleftarrow \mathcal{G}.\mathrm{Init}(pk, ctx)$
  $\alpha \leftarrow ctx;\ \textbf{ret}\ out$

$\mathbf{Final}(in)$
  $\textbf{ret}\ \mathcal{G}.\mathrm{Final}^{\mathcal{I}^b_{sk}}(st, in)$

$\mathbf{Call}(in)$
  $\textbf{ret}\ \mathcal{G}.\mathrm{Call}^{\mathcal{I}^b_{sk}}(\&st, in)$

$\mathbf{Op}(ctx, op, in)$
  $\textbf{if}\ ctx = \alpha\ \textbf{then ret}\ \bot$
  $\textbf{if}\ b = 1\ \textbf{then ret}\ \mathcal{I}^1_{sk}(ctx, op, in)$
  $\textbf{ret}\ \mathcal{S}.\mathrm{Op}^{\mathcal{I}^0_{sk}}(\&\sigma, ctx, op, in)$
```

Fig. 3. Top-left: The GAP1 experiment for interface $\mathcal{I}$, game $\mathcal{G}$, simulator $\mathcal{S}$, and adversary $\mathcal{D}$. Top-right: The GAP2 experiment for interfaces $\mathcal{I}^1$ and $\mathcal{I}^0$, $\mathcal{G}$, $\mathcal{S}$, and $\mathcal{D}$.

not a syntactic property of the game itself. This is because an experiment might execute the game differently; for example, instead of invoking the initiator before the caller, the experiment could invoke the caller with state $st = \varepsilon$ each time. This may sound silly, but we have not given a syntactic condition on games that excludes this execution semantics. Because all experiments will run the game in the same way, we silently extend this definition of regularity to all experiments in the remainder of the paper. In our analyses in Sections 5 and 6, we will prove SEC/I security with respect to regular games. This condition is sufficient for ensuring context sepparability between operations performed by the adversary via direct access to the interface and those performed by the game.

Indistinguishability variants. We note that our definitions of SEC/I and SEC advantage are not appropriate for every game. For example, $\mathcal{G}$ might be a bit-*guessing* game (e.g., IND-CCA) in which the initiator flips a coin and the finalizer interprets its input as the adversary's guess. In order to normalize the adversary's advantage in such games, we define the IND-SEC/I advantage of SEC/I-adversary $\mathcal{A}$ as $\mathbf{Adv}^{\text{ind-sec/i}}_{\mathcal{I},\mathcal{G}}(\mathcal{A}) = 2\mathbf{Adv}^{\text{sec/i}}_{\mathcal{I},\mathcal{G}}(\mathcal{A}) - 1$. (Similarly for IND-SEC.)

4.1 Simulatability of an Interface

Intuitively, the "gap" between the SEC/I and SEC security of an interface $\mathcal{I}$ with respect to game $\mathcal{G}$ is driven by any extra leverage the attacker gains by interacting with $\mathcal{I}$ directly. In this section, we formalize an experiment that aims to measure the size of this gap for a given $\mathcal{I}$ and $\mathcal{G}$. We also define a related experiment that measures the relative security "gap" between a pair of interfaces $(\mathcal{I}_1, \mathcal{I}_0)$ with respect to a given game. This is particularly useful when the operations permitted by $\mathcal{I}_1$ are a superset of those permitted by $\mathcal{I}_0$. For example, in Section 5, we will use this notion to analyze the change in security when operations are added to an existing interface. Both of these experiments will make use of simulators, so let us first define these.

Definition 4 (Simulators). A simulator $\mathcal{S}$ is a tuple of algorithms $(\mathrm{Init}, \mathrm{Op})$ defined as follows:

- $\mathrm{Init}(\mathbf{str}\, pk) \mapsto \mathbf{str}\, \sigma$. The *initiator* takes as input a public key and outputs the simulator's initial state σ.
- $\mathrm{Op}^{\mathcal{O}}(\&\mathbf{str}\, \sigma, \mathbf{str}\, ctx, op, in) \mapsto \mathbf{str}\, out$. The *operator* takes as input a reference to the simulator state (which it may update as a side-effect) and a triple of strings (ctx, op, in) and outputs a string out. Oracle $\mathcal{O}$ is an interface oracle defined just as for games.

In the remainder, we may denote $\mathcal{S}.\mathrm{Op}^{\mathcal{O}}(\&\sigma, ctx, op, in)$ by $\mathcal{S}^{\mathcal{O}}_{\sigma}(ctx, op, in)$. We say that $\mathcal{S}$ is (t, q_I)-resource if each algorithm is t-time and the caller makes at most q_I queries to its oracle. ♦

Definition 5 (GAP1/2 security). Figure 3 defines two experiments: GAP1 and GAP2. Each involves a simulator $\mathcal{S}$, an adversary $\mathcal{D}$, and a game $\mathcal{G}$; GAP1 involves a single interface $\mathcal{I}$, while GAP2 involves a pair interfaces $(\mathcal{I}_1, \mathcal{I}_0)$. Both begin by choosing a challenge bit b at random, executing the key generator ($\mathcal{I}$.Gen in GAP1 and $\mathcal{I}^b$.Gen in GAP2), and initializing the simulator via $\mathcal{S}$.Init on input of the public key. The adversary is then executed on input of the public key and with four oracles:

- **Init**, **Final**, and **Call** execute the game just like in the SEC/I experiment; interface queries are answered by $\mathcal{I}$.Op in GAP1 and $\mathcal{I}^b$.Op in GAP2.
- **Op** processes (ctx, op, in) as follows. If ctx is equal to the game context, then it returns $\perp$ (just as in SEC/I). If $b = 1$, then it returns $\mathcal{I}.\mathrm{Op}(sk, ctx, op, in)$ in GAP1 and $\mathcal{I}^1.\mathrm{Op}(sk, ctx, op, in)$ in GAP2; if $b = 0$, theni the oracle returns $\mathcal{S}.\mathrm{Op}^{\perp}(\&\sigma, ctx, op, in)$ in GAP1 and $\mathcal{S}.\mathrm{Op}^{\mathcal{I}^0_{sk}}(\&\sigma, ctx, op, in)$ in GAP2. (The "$\perp$" oracle given to $\mathcal{S}$ denotes the interface oracle that just returns $\perp$ on any query.)

The outcome of the experiment is the bit d output by $\mathcal{D}$ when it halts. A valid GAP1 (resp. GAP2) adversary makes a single query to **Init**, this being its first query; it may then make any number of queries to **Call** and **Op**. It completes its execution by making a single query to **Final**. We define the advantage of a (valid) GAP1-adversary $\mathcal{D}$ in attacking $\mathcal{I}$ with respect to $\mathcal{G}$ as

$$\mathbf{Adv}^{\mathrm{gap1}}_{\mathcal{I},\mathcal{G}}(\mathcal{S}, \mathcal{D}) = 2 \Pr\left[\, \mathbf{Exp}^{\mathrm{gap1}}_{\mathcal{I},\mathcal{G}}(\mathcal{S}, \mathcal{D}) \,\right] - 1 \,.$$

We call an GAP1 adversary (t, q_G, q_I)-resource if it is t-time and makes at most q_G and q_I queries to **Call** and **Op** respectively. We define the maximum advantage of any $\boldsymbol{r}$-resource GAP1 adversary (for a given $\mathcal{I}, \mathcal{G}, \mathcal{S}$) as $\mathbf{Adv}^{\mathrm{gap1}}_{\mathcal{I},\mathcal{G}}(\mathcal{S}, \boldsymbol{r})$.

Define $\mathbf{Adv}^{\mathrm{gap2}}_{\mathcal{I}^1,\mathcal{I}^0,\mathcal{G}}(\mathcal{S},\mathcal{D})$ and $\mathbf{Adv}^{\mathrm{gap2}}_{\mathcal{I}^1,\mathcal{I}^0,\mathcal{G}}(\mathcal{S},\boldsymbol{r})$ in kind. Informally, we say that $\mathcal{I}$ (resp. $(\mathcal{I}^1,\mathcal{I}^0)$) is GAP1 (resp. GAP2) secure for $\mathcal{G}$ if for every efficient GAP1 (resp. GAP2) adversary $\mathcal{D}$ there exists an efficient $\mathcal{S}$ such that $\mathcal{D}$ has small advantage.

Finally, we say that a simulator is *regular* for GAP1 (resp. GAP2) if each time it is called with input context ctx, each of its interface queries have the form (ctx, op, in) for some $op, in \in \{0,1\}^*$. ♦

4.2 The Composition Theorem

An interface $\mathcal{I}$ being GAP1 secure for $\mathcal{G}$ means that whatever information an SEC/I adversary learns in its attack against $\mathcal{G}$ it can (efficiently) compute on its own without interacting with the **Op** oracle. Thus, if $\mathcal{I}$ is both SEC and GAP1 secure for $\mathcal{G}$, then it should be that $\mathcal{I}$ is also SEC/I secure for $\mathcal{G}$. Relatedly, for any pair of interfaces $(\mathcal{I}^1,\mathcal{I}^0)$ and game $\mathcal{G}$, if $(\mathcal{I}^1,\mathcal{I}^0)$ is GAP2 secure for $\mathcal{G}$ and $\mathcal{I}^0$ is SEC/I secure for $\mathcal{G}$, then $\mathcal{I}^1$ is SEC/I-secure for $\mathcal{G}$, too. Theorem 1 makes these claims precise. To support upcoming results in Sections 5 and 6, we state and prove our composition theorem in the ROM. So, let us first formalize the ROM in our setting.

The ROM. When modeling a function $H : \mathcal{X} \to \mathcal{Y}$ as a random oracle (RO) in an experiment, we declare an associative array $\mathbf{elem}_{\mathcal{Y}}\,\pi[\,]$ and a set $\mathcal{Q}$ (initially empty) and define three oracles: **P**, **Q**, and **R**. The last of these is the usual RO: on input of $X \in \mathcal{X}$, oracle **R** checks to see if π_X is defined (i.e., $\pi_X \neq \diamond$); if not, then it samples π_X from $\mathcal{Y}$ according to the distribution induced on $\mathcal{Y}$ by H. (Usually $\mathcal{Y}$ will be finite and the distribution will be uniform.) Finally, it returns π_X. We call an algorithm *q_R-ro-bound* if it makes at most q_R queries to **R** during any execution; a game, interface, or simulator is q_R-ro-bound if each of its constituent algorithms is q_R-ro-bound. Experiments are lifted to the ROM by providing each named algorithm oracle access to **R**. In addition, each query X to **R** made by the adversary is added to the set $\mathcal{Q}$.

Just as we measure an adversary's runtime using the experiment in which it is executed, our convention will be that an adversary's RO-query budget accounts for all queries to **R** made by it or any other algorithm (including the simulator) during the course of the experiment. That is, XXX-adversary $\mathcal{A}$ is q_R-ro-bound if $\mathbf{Exp}^{\mathrm{xxx}}(\mathcal{A})$ is q_R-ro-bound. We say an algorithm is $(\boldsymbol{r} \,\|\, q_R)$-resource if it is $\boldsymbol{r}$-resource and q_R-ro-bound. (Note that $\boldsymbol{r} \,\|\, q_R$ is a tuple, since $\boldsymbol{r}$ is a tuple and q_R is a singleton.) Let $\psi : \{0,1\}^* \times \mathcal{X} \to \{0,1\}$ be a function. We say that a game $\mathcal{G}$ is *ψ-ro-regular* (for the associated experiment) if each of its RO queries $X \in \mathcal{X}$ satisfies $\psi(\alpha, X)$, where α is the game context used to initialize it in the experiment. Similarly, we say that an interface $\mathcal{I}$ is ψ-ro-regular if each of $\mathcal{I}$.Op's RO queries $X \in \mathcal{X}$ satisfies $\psi(ctx, X)$, where ctx is the provided context string.

The other two oracles (**P** and **Q**) are used to specify additional powers made available to simulators in security proofs. Oracle **P** takes as input a pair $(X,Y) \in \mathcal{X} \times \mathcal{Y}$ and sets $\pi[X] \leftarrow Y$, allowing the simulator to "program" the

RO. Oracle $\mathbf{Q}$ simply returns the set $\mathcal{Q}$ of RO queries made by the adversary so far, allowing the simulator to "observe" the adversary's RO queries as it makes them. We emphasize that $\mathbf{P}$ and $\mathbf{Q}$ formalize powers of the simulator that are usually left implicit, but are essential to certain proof techniques [16,13]. We introduce *oracle-relative* simulators as a means of formalizing the requirements of the simulator for composition.

Definition 6 (Oracle-relative simulators). Let $\mathcal{O}$ be an oracle in an experiment. An $\mathcal{O}$*-relative* simulator $\mathcal{S}$ is one for which both the initiator and operator expect oracle access to $\mathcal{O}$; we say that $\mathcal{S}$ is *c-*$\mathcal{O}$*-bound* if each algorithm makes at most c such queries on any execution. Let $\mathcal{X}$ and $\mathcal{Y}$ be sets and let $\mu_1, \mu_2 \geq 0$ be real numbers. In the ROM we say that a $\mathbf{P}$-relative simulator is (μ_1, μ_2)*-min-entropy* if for all $(X', Y') \in \mathcal{X} \times \mathcal{Y}$ and each query (X, Y) to $\mathbf{P}$, it holds that $\Pr\left[\, X = X' \,\right] \leq 2^{-\mu_1}$ and $\Pr\left[\, Y = Y' \,\right] \leq 2^{-\mu_2}$. ♦

Theorem 1. *Let $\mathcal{I}^1$ and $\mathcal{I}^0$ be interfaces, let $\mathcal{G}$ be a game, and let $H : \{0,1\}^* \to \{0,1\}^h$ be a function modeled as a random oracle. Let $q_G, q_I, q_R, t, c_I, c_R, c_P, s \geq 0$ be integers such that $s = O(t/(q_I+1))$, and let $\mu_1, \mu_2 \geq 0$ be real numbers such that $\mu_2 \leq h$. Let $\boldsymbol{r} = (t, q_G, q_I, q_R)$. Then, for every regular, $\mathbf{P}$- and $\mathbf{Q}$-relative simulator $\mathcal{S}$ that is (s, c_I, c_R)-resource, c_P-$\mathbf{P}$-bound, and (μ_1, μ_2)-min-entropy, it holds that*

(i) $\mathbf{Adv}^{\text{sec/i}}_{\mathcal{I}^1,\mathcal{G}}(\boldsymbol{r}) \leq \epsilon + \mathbf{Adv}^{\text{sec}}_{\mathcal{I}^1,\mathcal{G}}(O(t), q_G, \hat{q_R}) + \mathbf{Adv}^{\text{gap1}}_{\mathcal{I}^1,\mathcal{G}}(\mathcal{S}, \hat{\boldsymbol{r}})$ *and*

(ii) $\mathbf{Adv}^{\text{sec/i}}_{\mathcal{I}^1,\mathcal{G}}(\boldsymbol{r}) \leq \epsilon + \mathbf{Adv}^{\text{sec/i}}_{\mathcal{I}^0,\mathcal{G}}(O(t), q_G, c_I q_I, \hat{q_R}) + \mathbf{Adv}^{\text{gap2}}_{\mathcal{I}^1,\mathcal{I}^0,\mathcal{G}}(\mathcal{S}, \hat{\boldsymbol{r}})\,,$

where $\epsilon = (c_P q_I)(q_R/2^{\mu_1 - 1} + 2^{h-\mu_2} - 1)$, $\hat{q_R} = q_R + (c_R + c_P)(q_I + 1)$, and $\hat{\boldsymbol{r}} = (O(t), q_G, q_I, \hat{q_R})$.

We must defer the proof to the full version [29]. Except for accounting for the simulator's powers in the ROM, the proof is closely related to [33, Theorem 1]. A few observations about this result are in order. First, we note that the ϵ term in the bound is only non-zero for simulators that program the RO. Second, it is sufficient for the domain points programmed by the simulator to be high min-entropy, but the bound is vacuous unless the corresponding range points are essentially uniform (because of the $2^{h-\mu_2}$ term in the expression for ϵ). When the programmed domain points are high min-entropy, neither the game nor the GAP2 distinguisher is likely to call the RO on the domain points programmed by the simulator. This fact, and the uniformity of programmed range points, allows us to compose the GAP1/2 distinguisher and the simulator $\mathcal{S}$ into a new SEC/I adversary, despite the fact that $\mathcal{S}$ may program the RO, but the SEC/I adversary may not. Likewise, the simulator "observing" the distinguisher's RO queries is not an issue for this composition.

A necessary condition for Theorem 1(ii). Condition (ii) of the composition theorem characterizes a sufficient property of $(\mathcal{I}^1, \mathcal{I}^0)$ and $\mathcal{G}$ such that it is safe to replace $\mathcal{I}^0$ with $\mathcal{I}^1$ (GAP2). This tells us, in particular, what sorts of operations are safe to expose in an API without breaking applications. We would also like a characterization of what sorts of operations are *not* safe, i.e., a necessary condition for Theorem 1(ii). We find that if wGAP2 security (defined below) does not hold for $(\mathcal{I}^1, \mathcal{I}^0)$, then there are games $\mathcal{G}$ for which $\mathcal{I}^1$ is not SEC/I secure, even if $\mathcal{I}^0$ is SEC/I secure for $\mathcal{G}$ (Theorem 2). We will use this result to rule out certain API-design choices in the remainder of the paper.

Definition 7 (wGAP2 security). The wGAP2 experiment is defined much like GAP2, except it does not involve a game. (Pseudocode for this definition is provided in the full version [29].) A wGAP2 adversary takes as input a string and outputs a bit and expects access to an interface oracle. Let $\mathcal{I}^1$ and $\mathcal{I}^0$ be interfaces, $\mathcal{S}$ be a simulator, and $\mathcal{D}$ be a wGAP2 adversary. The wGAP2 experiment for $(\mathcal{I}^1, \mathcal{I}^0)$, $\mathcal{S}$, and $\mathcal{D}$, denoted $\mathbf{Exp}^{\mathrm{wgap2}}_{\mathcal{I}^1,\mathcal{I}^0}(\mathcal{S}, \mathcal{D})$, is defined just like the GAP2 experiment in Figure 3, except that $\mathcal{D}$ is only executed with access to oracle **Op**, and since there is no game context, we remove line 3:20. Define the advantage of $\mathcal{D}$ in distinguishing $\mathcal{I}^1$ from $\mathcal{I}^0$ with respect to simulator $\mathcal{S}$ as $\mathbf{Adv}^{\mathrm{wgap2}}_{\mathcal{I}^1,\mathcal{I}^0}(\mathcal{S}, \mathcal{D}) = 2\Pr\left[\,\mathbf{Exp}^{\mathrm{wgap2}}_{\mathcal{I}^1,\mathcal{I}^0}(\mathcal{S}, \mathcal{D})\,\right] - 1$. Informally, we say that $(\mathcal{I}^1, \mathcal{I}^0)$ is wGAP2 secure if for every efficient adversary $\mathcal{D}$, there is an efficient simulator $\mathcal{S}$ such that $\mathcal{D}$'s advantage is small. We say $\mathcal{D}$ is (t, q_I)-resource if it is t-time and makes at most q_I queries to **Op**. ♦

Theorem 2 (wGAP2 is necessary for Theorem 1(ii)). *Let $\mathcal{I}^1$ and $\mathcal{I}^0$ be interfaces, let $\mathcal{B}$ be an SEC/I adversary, and let $\mathcal{D}$ be a wGAP2 adversary. There exist a game $\mathcal{G}$, SEC/I-adversary $\mathcal{A}$, and simulator $\mathcal{S}$ such that*

$$\mathbf{Adv}^{\mathrm{wgap2}}_{\mathcal{I}^1,\mathcal{I}^0}(\mathcal{S}, \mathcal{D}) + \mathbf{Adv}^{\mathrm{sec/i}}_{\mathcal{I}^0,\mathcal{G}}(\mathcal{B}) \le \mathbf{Adv}^{\mathrm{sec/i}}_{\mathcal{I}^1,\mathcal{G}}(\mathcal{A})\,.$$

Moreover, if $\mathcal{D}$ is (s, r)-resource, $\mathcal{B}$ is (t, q_G, q_I)-resource, and $t = O(s)$, then $\mathcal{A}$ is $(O(t), q_G, q_I + r)$-resource and $\mathcal{S}$ is $(t, 1)$-resource.

Note that this result is easily lifted to the ROM. The proof (provided in the full version [29]) is in the same spirit as that of [24, Theorem 1], but there are some subtleties. The crux of the argument, which was adapted from Maurer, Renner, and Holenstein [24], is that the game $\mathcal{G}$ is defined using the adversary $\mathcal{D}$ so that the winning condition depends on $\mathcal{D}$ doing something "bad" (in particular, outputting 1). This allows us to relate $\mathcal{B}$'s advantage to $\mathcal{D}$'s. (We remark on the necessity of GAP1 itself for composition in the full version [29].)

5 Discrete Log Interfaces

In this section we bring our framework to bear on a few common operations for discrete log (DL) interfaces. We first recall some standard definitions from the cryptographic literature and formally define *DL* interfaces and *signing* interfaces.

Preliminaries. Refer to the CDH and GDH experiments in Figure 4. Define the advantage of an adversary $\mathcal{A}$ in solving an instance of the *computational DH (CDH)* problem for $\mathbb{G}$ as $\mathbf{Adv}^{\mathrm{cdh}}_{\mathbb{G}}(\mathcal{A}) = \Pr[\mathbf{Exp}^{\mathrm{cdh}}_{\mathbb{G}}(\mathcal{A})]$ and let $\mathbf{Adv}^{\mathrm{cdh}}_{\mathbb{G}}(t)$ denote the maximum advantage of any t-time CDH-adversary. Define the advantage of an adversary $\mathcal{A}$ in solving an instance of the *gap DH (GDH)* problem [26] for $\mathbb{G}$

```
$\mathbf{Exp}^{\text{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A})$
1  $\textbf{dec}\ X, Z \in \mathbb{G};\ y \in \mathbb{Z}_n$
2  $(\underline{X}, sk) \twoheadleftarrow \mathcal{I}.\text{Gen}(\,)$
3  $y \twoheadleftarrow \mathbb{Z}_n$
4  $Z \twoheadleftarrow \mathcal{A}^{\mathcal{I}_{sk}}(X, yG)$
5  $\textbf{ret}\ (Z = yX)$

$\mathbf{Exp}^{\text{cdh}}_{\mathbb{G}}(\mathcal{A})$ / $\boxed{\mathbf{Exp}^{\text{gdh}}_{\mathbb{G}}(\mathcal{A})}$
6  $x, y \twoheadleftarrow \mathbb{Z}_n$
7  $Z \twoheadleftarrow \mathcal{A}(xG, yG)$
8  $\boxed{Z \twoheadleftarrow \mathcal{A}^{\mathbf{DDH}}(xG, yG)}$
9  $\textbf{ret}\ (Z = xyG)$

$\mathbf{DDH}(A, B, C)$
10  $a \leftarrow \log_G A$
11  $b \leftarrow \log_G B$
12  $c \leftarrow \log_G C$
13  $\textbf{ret}\ (c = ab)$
```

Fig. 4. Let $\mathbb{G} = \langle G \rangle$ be a represented, additive group of order n and let $\mathcal{I}$ be a DL interface for $\mathbb{G}$. Left: IDH problem for $(\mathbb{G}, \mathcal{I})$. Right: CDH and GDH problems for $\mathbb{G}$.

as $\mathbf{Adv}^{\text{gdh}}_{\mathbb{G}}(\mathcal{A}) = \Pr[\mathbf{Exp}^{\text{gdh}}_{\mathbb{G}}(\mathcal{A})]$. Depending on the group $\mathbb{G}$ and the model of computation, it may not be possible to evaluate $\mathcal{A}$'s **DDH** queries efficiently; for the purpose of accounting for $\mathcal{A}$'s resources, we will regard the discrete log computations on lines 4:7–8 as constant time operations. Let $\mathbf{Adv}^{\text{gdh}}_{\mathbb{G}}(t, q)$ denote the maximum advantage of any t-time GDH-adversary that makes at most q queries to its **DDH** oracle. Informally, we say CDH (resp. CDH) is hard for $\mathbb{G}$ if the CDH (resp. GDH) advantage of any efficient adversary is small.

Define the CR advantage of an adversary $\mathcal{C}(\,) \mapsto \mathbf{elem}_{\mathcal{X}\times\mathcal{X}}$ in finding collisions for function $H : \mathcal{X} \to \mathcal{Y}$ as $\mathbf{Adv}^{\text{cr}}_{H}(\mathcal{C}) = \Pr\big[\, X \neq Y \wedge H(X) = H(Y) : (X, Y) \twoheadleftarrow \mathcal{C}(\,) \,\big]$.

Definition 8 (DL and signing interfaces). Let $\mathbb{G} = \langle G \rangle$ be a represented, additive group of order n. A DL interface for $\mathbb{G}$ is an interface $\mathcal{I}$ with an associated *scalar computer*, a deterministic algorithm $\text{Scal}(\textbf{str}\ sk) \mapsto \textbf{int}\ s$ such that for every $(pk, sk) \in [\mathcal{I}.\text{Gen}(\,)]$ it holds that $pk = \underline{sG}$, where $s = \mathcal{I}.\text{Scal}(sk)$. We say that $\mathcal{I}$ is *simple* if $\mathcal{I}.\text{Scal}(sk) = s$ just in case $sk = \underline{s}$.

A *signing interface* is an interface $\mathcal{DS}$ with an associated deterministic algorithm $\mathcal{DS}.\text{Verify}(\textbf{str}\ pk, ctx, M, T) \mapsto \textbf{bool}\ v$, called the *verifier*, for which $T \in [\mathcal{DS}(sk, ctx, \mathsf{sig}, M)]$ iff $\mathcal{DS}.\text{Verify}(pk, ctx, M, T) = 1$ for all $ctx, M, T \in \{0,1\}^*$ and $(pk, sk) \in [\mathcal{DS}.\text{Gen}(\,)]$. (This is analogous to the correctness condition for standard signature schemes.) We may denote $\mathcal{DS}.\text{Op}(sk, ctx, \mathsf{sig}, M)$ by $\mathcal{DS}.\text{Sign}(sk, ctx, M)$ and refer to $\mathcal{DS}.\text{Sign}$ as the *signer*. We say that a game is *$\mathcal{DS}$-regular* (for the associated experiment) if each time it invokes $\mathcal{DS}.\text{Verify}$, it does so on input of (pk, α, M, T), where α is the game context used to initialize it and $pk, M, T \in \{0,1\}^*$. ♦

5.1 Diffie-Hellman

Let $\mathbb{G} = \langle G \rangle$ be an additive, represented group of order n. Let $\mathcal{I}$ be a DL interface for $\mathbb{G}$ and define $\mathcal{I}_{+\text{dh}}$ as the pair of algorithms $(\mathcal{I}.\text{Gen}, \text{Op})$, where Op is defined as follows. On input of (sk, ctx, op, in), if $op = \mathsf{dh}$ and $Q \in \mathbb{G}$, where Q is the element of $\mathbb{G} \cup \{\diamond\}$ encoded by in, then return $\underline{sQ}$, where $s = \mathcal{I}.\text{Scal}(sk)$; otherwise return $\mathcal{I}(sk, ctx, op, in)$. We refer to dh as the *DH operator*. (Note that point validation [22] for this operation is implicitly enforced by our conventions for represented groups; see Section 2.)

It is well known that exposing such a "static DH oracle" is not generally secure [11], but its practical impact on security can be subtle, and its presence in an interface is often hard to recognize [2,12]. In order to rule out the security of exposing the DH operation (inadvertently or not), we formalize a property of $\mathcal{I}$ that, if it holds, implies that $(\mathcal{I}_{+\mathrm{dh}}, \mathcal{I})$ is wGAP2 *insecure*; by Theorem 2, this implies that $\mathcal{I}_{+\mathrm{dh}}$ is not SEC/I secure in general. We then build on this result by considering whether it is safe to expose some function of the output (e.g., a hash or key-derivation function); when we model the function as a random oracle, we find that this is not wGAP2 secure.

Insecurity of exposing DH easily follows from the hardness of a variant of the CDH problem for $\mathbb{G}$ associated with $\mathcal{I}$. The *interface-relative DH (IDH)* problem for $(\mathbb{G}, \mathcal{I})$ is as follows.

Definition 9 (The IDH problem). Refer to the IDH experiment for $\mathbb{G}$ and $\mathcal{I}$ in Figure 4. The experiment first runs $\mathcal{I}$.Gen to get the public key X and secret key sk. It then chooses a random $y \in \mathbb{Z}_n$ and runs the adversary $\mathcal{A}$ on input of (X, yG) and with oracle access to $\mathcal{I}_{sk}$; the adversary wins if it outputs yX. Define the advantage of IDH-adversary $\mathcal{A}$ as $\mathbf{Adv}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A}) = \Pr\left[\mathbf{Exp}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A})\right]$. An IDH adversary is (t, q)-resource if it is t-time and makes at most q queries to its interface oracle; as usual, we denote the maximum advantage of any $\boldsymbol{r}$-resource IDH adversary by $\mathbf{Adv}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\boldsymbol{r})$. Informally, we say the IDH problem is hard for $(\mathbb{G}, \mathcal{I})$ if $\mathbf{Adv}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A})$ is small for every efficient $\mathcal{A}$. ⧫

We will use this problem as a sort of litmus test to rule out insecure API designs. In Section 5.2 we show (via Theorem 1(i)) that CDH and IDH are equivalent relative to EdDSA, and in in Section 6 we show that GDH and IDH are equivalent relative to Noise. To prove that hardness of the IDH problem for $(\mathbb{G}, \mathcal{I})$ implies the wGAP2 insecurity of $(\mathcal{I}_{+\mathrm{dh}}, \mathcal{I})$, we exhibit a wGAP2 adversary $\mathcal{D}$ such that in order for any simulator $\mathcal{S}$ to thwart $\mathcal{D}$, it must solve an instance of IDH for $(\mathbb{G}, \mathcal{I})$.

Theorem 3. *Suppose that n is prime and let $t, q_I \geq 0$ be integers. There is a $(O(t), 1)$-resource wGAP2-adversary $\mathcal{D}$ such that for all (t, q_I)-resource $\mathcal{S}$, there is a $(O(t), q_I)$-resource IDH-adversary $\mathcal{A}$ such that $\mathbf{Adv}^{\mathrm{wgap2}}_{\mathcal{I}_{+\mathrm{dh}},\mathcal{I}}(\mathcal{S}, \mathcal{D}) = 1 - \mathbf{Adv}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A})$.*

Proof. Define adversary $\mathcal{D}^{\mathbf{Op}}(\underline{P})$ as follows. First run $r \twoheadleftarrow \mathbb{Z}_n^*$, then ask $\underline{Z} \twoheadleftarrow \mathbf{Op}(\varepsilon, \mathsf{dh}, \underline{rG})$. If $r^{-1}Z = P$, then return 1; otherwise return 0. Let d_{b1} denote the probability that $\mathcal{D}$ outputs 1 conditioned on the event that its challenge bit is b. First, if $b = 1$, then the response to $\mathcal{D}$'s query will be $Z = srG$, where $P = sG$. Since n is prime, r has a unique inverse $1/r \pmod n$, and so $r^{-1}Z = r^{-1}srG = sG = P$. It follows that $d_{11} = 1$. Now consider the probability that $r^{-1}Z = P$ given that $b = 0$ and define adversary $\mathcal{A}^{\mathcal{O}}(P, Q)$ as follows. It first executes $\sigma \twoheadleftarrow \mathcal{S}.\mathrm{Init}(\underline{P})$, then $\underline{Z} \twoheadleftarrow \mathcal{S}^{\mathcal{O}}(\&\sigma, \varepsilon, \mathsf{dh}, \underline{Q})$. Finally, it returns Z. Then the probability that $\mathcal{A}$ wins is precisely the probability that, in $\mathcal{D}$'s game, simulator $\mathcal{S}$ outputs $\underline{Z}$ such that $r^{-1}Z = P \iff rP = Z$, and so $d_{01} = \Pr\left[\,\mathbf{Exp}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A})\,\right]$. □

Functional DH. Many applications do not make direct use of static DH, but some function of its output. In particular, it is common to apply a hash or key-derivation function to the shared secret, perhaps binding it to some context, e.g., the transcript hash in TLS or, as we will see, the CipherState in Noise. Therefore, it is worth considering whether exposing this intermediate functionality is secure.

Let $\mathcal{F} : \mathbb{G} \times \{0,1\}^* \to \{0,1\}^h$ be a function. Define the interface $\mathcal{I}_{+\mathrm{fdh}}$ as the pair of algorithms $(\mathcal{I}.\mathrm{Gen}, \mathrm{Op})$, where Op is defined as follows. On input of (sk, ctx, op, in), if $op = \mathsf{fdh}$ and $Q \in \mathbb{G}$, where Q is the element of $\mathbb{G} \cup \{\diamond\}$ encoded by in, then return $\mathcal{F}(sQ, ctx)$; otherwise return $\mathcal{I}.\mathrm{Op}(sk, ctx, op, in)$. We call $op = \mathsf{fdh}$ the *functional DH operator*.

Exposing functional DH is also wGAP2 insecure. The proof is more involved, but follows similar lines as Theorem 3. We cannot directly exploit the algebraic structure of the DH operator as we did above, since rather than getting $\underline{sQ}$ in response to its query, adversary $\mathcal{D}$ gets $\mathcal{F}(sQ, ctx)$. Instead, we model $\mathcal{F}$ as a random oracle and hope that the simulator manages to query the oracle with the correct point. We prove the following in the full version [29]:

Theorem 4. *Suppose that n is prime and let $t, q_I, q_R \geq 0$ be integers. When $\mathcal{F}$ is modeled as a random oracle, there is a $(O(t), 1, 1)$-resource wGAP2-adversary $\mathcal{D}$ such that for all (t, q_I, q)-resource, $\mathbf{P}$- and $\mathbf{Q}$-relative, and p-$\mathbf{P}$-bound $\mathcal{S}$, there is a $(O(t+q), q_I)$-resource IDH-adversary $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\mathrm{wgap2}}_{\mathcal{I}_{+\mathrm{fdh}},\mathcal{I}}(\mathcal{S},\mathcal{D}) + \epsilon \geq 1 - \mathbf{Adv}^{\mathrm{idh}}_{\mathbb{G},\mathcal{I}}(\mathcal{A})\,,$$

where $\mathcal{I}$ is 0-ro-bound, $\epsilon = \hat{q}/n + \hat{q}^2/2^{h-1}$, and $\hat{q} = 2(q+p)$.

Discussion. The existence of a static DH oracle in an interface can be difficult to recognize, and its impact on security is often quite subtle. Acar, Ngyuen, and Zavarucha [2] discovered that an early version of the TPM standard exposed such an oracle via flexible API calls designed to support a wide variety of protocols. Indeed, a rigorous analysis of the standard in our attack model would have unearthed this subtlety. It would be worthwhile to study the proposal of Camenish et al. [12], which aims to remove the TPM oracle while still supporting a large variety of useful applications. More generally, we suggest that the approach developed in this paper could be used to vet API standards before they are implemented to help uncover such flaws. Though the problem with TPM was obvious in hindsight, it is possible that more flaws lurk in this and other API designs.

5.2 EdDSA

Unlike signature schemes like RSA-PSS or ECDSA, the standardized version of EdDSA (RFC 8032 [17]) admits variants that are context separable, allowing us

```
Gen( )
    $K \twoheadleftarrow \{0,1\}^b$; $s \leftarrow \mathrm{Scal}(K)$
    ret $(\underline{sG}, K)$

Verify$(pk, ctx, M, T)$
    dec $P, R \in \mathbb{G}$; $x, t \in \mathbb{N}$
    $\underline{P} \leftarrow pk$; $\underline{R, x} \leftarrow T$
    if $\neg R \vee \neg x$ then ret 0
    $\underline{t}_{2b} \leftarrow \mathcal{H}(vr(ctx) \,\|\, \underline{R} \,\|\, \underline{P} \,\|\, ph(M))$
    ret $\left(x2^cG = 2^cR + t2^cP\right)$

Scal$(K)$
    ret $cl(\mathcal{H}(K)[:b])$

Sign$(K, ctx, M)$
    dec $r, t \in \mathbb{N}$
    $s \leftarrow \mathrm{Scal}(K)$; $X \leftarrow \mathcal{H}(K)[b+1:]$
    $\underline{r}_{2b} \leftarrow \mathcal{H}(vr(ctx) \,\|\, X \,\|\, ph(M))$
    $\underline{t}_{2b} \leftarrow \mathcal{H}(vr(ctx) \,\|\, \underline{rG} \,\|\, \underline{sG} \,\|\, ph(M))$
    $x \leftarrow r + st \pmod{n}$
    ret $\underline{rG, x}$
```

Fig. 5. Signing/DL interface $\mathcal{ED}$ for EdDSA. Let $b, c \in \mathbb{N}$ and let $\mathbb{G} = \langle G \rangle$ be a represented, additive group of order n. Let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{2b}$, $cl : \{0,1\}^b \to \mathbb{Z}_n \setminus \{0\}$, and $vr, ph : \{0,1\}^* \to \{0,1\}^*$ be functions.

to prove it GAP1 secure (in the ROM) for any game in which all signing and verifying operations are regular (Def. 8). We also show that any variant can be securely composed with any simple DL interface. After presenting our results, we will make the case for designing and deploying context-separable signatures in practice.

The standard specifies two concrete instantiations of EdDSA: Ed22519 and Ed448, whose names indicate the underlying group. The signing interface $\mathcal{ED}$ defined in Figure 5 specifies generic EdDSA; a concrete scheme is instantiated by selecting the group $\mathbb{G}$, integers b and c, and functions $\mathcal{H}$, cl, vr, and ph. The group is determined by a prime number $p > 2$, parameters for a (twisted) Edwards curve E (see [8, Section 2]), and a generator G of a prime order subgroup of $E(\mathbb{F}_p)$, where $E(\mathbb{F}_p)$ denotes the group of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ that lie on the curve E, and $\mathbb{F}_p$ denotes the finite field of order p. Define b so that $2^{b-1} > p$ and define c so that $\#E(\mathbb{F}_p) = n2^c$ (i.e., 2^c is the cofactor of $\mathbb{G}$). This choice of b makes it possible to encode signatures with $2b$ bits, and this choice of c is intended to mitigate small subgroup attacks [22]. The "clamping" function cl is similarly tailored to the underlying group: for Ed25519 and its variants, this function clears the first 3 bits, sets the second to last bit, and clears the last bit. (This ensures that $s = 2^{254} + 8x$ for a uniform random $x \in \mathbb{Z}_{2^{251}}$.) Finally, the algorithm variant is determined by the functions vr and ph. For example, the most common Ed25519 variant is obtained by setting $vr(X) = \varepsilon$ and $ph(X) = X$ for all X, but the standard also specifies variants that permit context (Ed25519ctx) and pre-hashing of the message (Ed25519ph). To provide context separability, the function vr must be collision resistant.

We begin our analysis by proving that the context-separable variants of EdDSA are GAP1 secure in the ROM for games in which the signing and verifying operations are regular (Theorem 5). The upcoming Corollary 1, which follows from Theorem 1(1) and Theorem 5, combined with the straightforward result that IDH implies CDH, gives a qualitative equivalence between CDH and IDH in terms of the security of (any variant of) EdDSA. We will then show that exposing *any* variant of EdDSA in any *simple* DL interface is GAP2 secure in general (Theorem 6). Fix EdDSA parameters $(\mathbb{G}, \mathcal{H}, cl, vr, ph, b, c)$ and let $\mathcal{ED}$ be the signing interface instantiated with these parameters as specified in Figure 5. Let $n = |\mathbb{G}|$.

Theorem 5. *Let $\mathcal{G}$ be an ($\mathcal{ED}$-)regular game and suppose that $n \leq 2^{b-1}$. When $\mathcal{H}$ is modeled as a random oracle, there exists a regular, $\mathbf{P}$-relative simulator $\mathcal{S}$ such that for all $t, q_G, q_I, q_R, c \geq 0$ there exists a $O(t + q_R q_I)$-time CR-adversary $\mathcal{C}$ such that $\mathbf{Adv}^{\text{gap1}}_{\mathcal{ED},\mathcal{G}}(\mathcal{S}, \boldsymbol{r}) \leq 2cq_I \mathbf{Adv}^{\text{cr}}_{vr}(\mathcal{C}) + 6q_R q_I/n$, where $\mathcal{G}$ is c-bound, $\mathcal{S}$ is $(\log n/2, 2b)$-min-entropy, $(O(t/(q_I + 1)), 1, 0)$-resource, and 1-$\mathbf{P}$-bound, and $\boldsymbol{r} = (t, q_G, q_I, q_R)$.*

We only give the high level idea of the a argument here; refer to the full version for the proof [29]. The simulator programs the random oracle with valid EdDSA signatures in the usual way (cf. [13, Section 4.4]). We must ensure, however, that signatures programmed by the simulator cannot be used by the adversary in an attack against the game $\mathcal{G}$. To do so, we use the collision resistance of vr to bound the probability that any interface query made via **Call** coincides with an interface query made via **Op**. For this argument to work, we must require that $\mathcal{G}$ is ($\mathcal{ED}$-)regular.

If the game in Theorem 5 makes no interface queries (i.e., is 0-bound), then CR security of vr is not required. This allows us to prove equivalence of IDH and CDH regardless of how vr is realized. The following corollary follows almost immediately from Theorems 1(i) and 5.

Corollary 1. *Let $r = |\text{Rng}\, cl|$ and suppose that $r \mid 2^b$ and $n \leq 2^{b-1}$. Then for all $t, q_I, q_R \geq 0$ it holds that $\mathbf{Adv}^{\text{idh}}_{\mathbb{G},\mathcal{ED}}(t, q_I, q_R) \leq n/r \mathbf{Adv}^{\text{cdh}}_{\mathbb{G}}(O(t + \hat{q})) + 7q_R q_I/n$, where $\mathcal{H}$ is modeled as a random oracle and $\hat{q} = q_R + q_I + 1$.*

The IDH experiment is equivalent to the SEC/I experiment with $\mathcal{ED}$ and a game $\mathcal{G}^{\text{cdh}}$ that specifies the CDH problem with one of the inputs being the public key provided to the game as input. We reduce the SEC security of $\mathcal{G}^{\text{cdh}}$ to SEC/I via Theorem 1(i) with help of the simulator exhibited in Theorem 5. Note that $\mathcal{G}^{\text{cdh}}$ is 0-ro-bound, and so CR security of vr does not arise in the bound. The result is obtained by observing that the SEC experiment for $\mathcal{ED}$ and $\mathcal{G}^{\text{cdh}}$ is essentially the CDH experiment for $\mathbb{G}$ modulo the distribution on the first input induced by $\mathcal{ED}$.Gen, which accounts for the n/r term. We refer the reader to the full version for the complete proof [29].

Finally, we show that EdDSA can be composed with any *simple* DL interface $\mathcal{I}$ without affecting the security of $\mathcal{I}$'s intended application. Let $\mathcal{I}$ be a simple DL interface for $\mathbb{G}$. We define a new interface $\mathcal{ED}_{+\mathcal{I}} = (\mathcal{ED}.\text{Gen}, \text{Op})$, where on input of (sk, ctx, op, in), algorithm Op returns $\mathcal{ED}.\text{Sign}(sk, ctx, in)$ if $op = \mathsf{sig}$ and returns $\mathcal{I}.\text{Op}(\underline{s}, ctx, op, in)$ otherwise, where $s = \mathcal{ED}.\text{Scal}(sk)$.

Theorem 6. *Let $\mathcal{G}$ be a game and suppose that $n \leq 2^{b-1}$. When $\mathcal{H}$ is modeled as a random oracle, there exists a regular, $\mathbf{P}$-relative simulator $\mathcal{S}$ such that for all $t, q_G, q_I, q_R \geq 0$ it holds that $\mathbf{Adv}^{\text{gap2}}_{\mathcal{ED}_{+\mathcal{I}},\mathcal{I},\mathcal{G}}(\mathcal{S}, \boldsymbol{r}) \leq 7q_R q_I/n$, where $\mathcal{S}$*

is $(\log n/2, 2b)$*-min-entropy,* $(O(t/(q_I + 1)), 1, 0)$*-resource, and* 1*-*$\mathbf{P}$*-bound, and* $\boldsymbol{r} = (t, q_G, q_I, q_R)$*.*

The restriction to simple interfaces is so that we can achieve context separation in the proof without using collision resistance of vr. The argument leverages the fact that $\mathcal{I}$ does not make use of the string X computed by the signer. Otherwise the proof is closely related to Theorem 5; we defer the details to the full version of this paper [29].

Discussion. The restrictions imposed on the game in Theorem 5 and the interface in Theorem 6 are very mild, but are required for context separability. If the game encodes the UF-CMA security of $\mathcal{ED}$, then this ensures that a signature generated via the interface cannot be used as a forgery in the game. But this "attack" is rather uninteresting and is only an artifact of our model. On the other hand, the game might specify the use of a signature scheme in a complex protocol like TLS in which digital signatures have a variety of uses, including client and server authentication and delegation of credentials for terminating TLS on a party's behalf [4]. In each of these cases the protocol binds the signature to a unique context string identifying its use (e.g., [32, Section 4.4.3]). Our abstraction boundary makes the requirements for such applications explicit. Because Ed25519ctx and Ed448ctx are context separable, Theorem 5 makes clear the conditions under which these algorithms are secure for their intended application, no matter how else they are used: the implementer must ensure that (1) the interface enforces context separation, and (2) signing/verification operations in the application always use the context that identify the application. We believe that exploiting this property of context-separable signatures would reduce the inherent complexity of designing and deploying protocols. (Indeed, it is also not difficult to design signature schemes to have this property.)

6 Noise

In this section we consider the GAP1 security of Noise [30], a framework for designing DL-based, two-party protocols. Noise provides a set of rules for processing *handshake patterns*, which define the sequence of interactions between an *initiator* and *responder* in a protocol. The processing rules involve three primitives: Diffie-Hellman (DH), an AEAD scheme, and a hash function. Each message sent or received by a host updates the host's state, which consists of the host's *ephemeral* (i.e., short-lived) and *static* (long-lived) secret keys, the peer's ephemeral and static public keys, shared state used to derive the symmetric key and associated data, the current symmetric key, and the current nonce. The symmetric key, nonce, and associated data are used to encrypt *payloads* accompanying each message, providing implicit authentication of a peer via confirmation of knowledge of their static secret.

Noise admits a wide variety of protocols. The processing rules are designed to make it easy to verify properites of handshake patterns, and considerable effort has gone into their formal analysis [14,19,23]. But the study of handshake patterns in isolation does not fully address the complexity of using Noise to build and deploy protocols. In practice, it is often necessary for the communicants to negotiate the details of the handshake, including the pattern, primitives, and cryptographic artifacts such as static keys and their certificates. All of this is out of scope of the core Noise specification, which aims to be as rigid as possible. As a result, there is an apparent gap between our understanding of the security that Noise provides and how it might be used in practice. One question that arises, which we will address here, is whether it is safe to reuse a single static key in many patterns.

We cast the Noise framework as an interface that exposes a host's static key for use in Noise protocols. The interface specifies how the host consumes (resp. produces) messages sent by (resp. to send to) the peer, and how its handshake state is updated as a side-effect. In other words, it implements the processing rules such that Noise patterns can be executed by making calls to the interface. Our goal is to prove GAP1 security with respect to the largest possible set of games, which would provide two benefits in practice. First and foremeost, it would imply joint security (up to context separation) of all patterns the interface implements; second it would provide a degree of robustness to cross protocol attacks by ensuring that, as long as context separation is enforced, vulnerabilities in one application cannot creep into another.

Our analysis sheds light on two limitations of Noise with respect to our security notions. The first is that *some* handshake patterns, if implemented by our interface, would allow for GAP1 attacks. We provide a formal characterization of the actions that give rise to these attacks, and we prove GAP1 security of our interface when they are excluded. The second issue is more subtle. To prove GAP1 security with respect to games in which the adversary may compromise the handshake state—for example, when modeling forward secrecy—it is necessary to tweak the Noise spec slightly. The processing rules explicitly bind the protocol context (i.e., a string that uniquely defines the handshake pattern and parameters) to the initial state of the protocol. While this provides a certain degree of context separability, the lack of binding to each state update precludes a proof of security relative to such games. We propose a simple and efficient modification of the processing rules that ensures context separability under these conditions, allowing us to prove security under minimal (and natural) assumptions about the game.

Of course, a consequence of these restrictions is that our analysis leaves open the security of key reuse in Noise *as it is*. In the full version of this paper [29], we will discuss what our results mean for Noise in practice and suggest directions for future work.

Preliminaries. Our analysis will use the standard notion of ciphertext integrity of AEAD schemes. A scheme for *authenticated encryption with associated data (AEAD)* is a pair of deterministic algorithms $\mathcal{AE} = (\text{Enc}, \text{Dec})$. The first, $\text{Enc}(\textbf{str}\, K, N, A, M) \mapsto \textbf{str}\, C$, maps a key K, nonce N, associated data A, and plaintext M to a ciphertext C. The second, $\text{Dec}(\textbf{str}\, K, N, A, C) \mapsto \textbf{str}\, M$, maps K, N, A, and C to M. We respectively define the key, nonce, associated-

NN:	NK:	NX:	IKpsk2:
$\rightarrow$ e,	$\leftarrow$ s	$\rightarrow$ e	$\leftarrow$ s
$\leftarrow$ e, ee	...	$\leftarrow$ e, ee, s, es	...
	$\rightarrow$ e, es		$\rightarrow$ e, es, s, ss
	$\leftarrow$ e, ee		$\leftarrow$ e, ee, se, psk

Fig. 6. Examples of Noise handshake patterns.

data (AD), and message space as the sets $\mathcal{K}, \mathcal{N}, \mathcal{A}, \mathcal{M} \subseteq \{0,1\}^*$ for which $\mathrm{Enc}(K, N, A, M) \neq \bot$ if and only if $(K, N, A, M) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$; correctness requires that $\mathrm{Dec}(K, K, N, A, \mathrm{Enc}(K, N, A, M)) = M$ for every such (K, N, A, M). (This condition implies that $\mathcal{AE}$ is both *correct* and *tidy* in the sense of Namprempre, Rogaway, and Shrimpton [25].) We say that $\mathcal{AE}$ has key-length k if $\mathcal{K} = \{0,1\}^k$ and nonce-length n if $\mathcal{N} = \{0,1\}^n$. We will use the standard notion of ciphertext integrity (INT-CTXT) for AEAD schemes in the presence of nonce-respecting adversaries; refer to the full version [29] for its precise definition. Define the advantage of an adversary $\mathcal{A}$ in breaking the ciphertext integrity of $\mathcal{AE}$ as $\mathbf{Adv}^{\text{int-ctxt}}_{\mathcal{AE}}(\mathcal{A}) = \Pr\left[\mathbf{Exp}^{\text{int-ctxt}}_{\mathcal{AE}}(\mathcal{A})\right]$. Let $\mathbf{Adv}^{\text{int-ctxt}}_{\mathcal{AE}}(t, q_E, q_D)$ denote the maximum advantage of any t-time adversary making at most q_E (resp. q_D) queries to **Enc** (resp. **Dec**).

6.1 Handshake and Message Patterns

By way of eliciting the formal tools we will need in our analysis, we begin this section with a brief overview of how handshake patterns are specified. Figure 6 recalls four patterns from the standard [30]. The first, referred to as the "NN" pattern, encodes an unauthenticated DH key exchange as a sequence of *handshake messages*, which in turn encode sequences of *tokens*. In the first message ($\rightarrow$ e) the initiator generates an ephemeral DH key pair and sends the public key to the responder. In the next handshake message ($\leftarrow$ e, ee), the responder generates an ephemeral key pair (e), computes the DH shared secret and derives a symmetric key (ee), then sends the ephemeral public key in its response. Every message includes a possibly AEAD-encrypted *payload*. Encryption is opportunistic. Once a shared secret is established, everything that can be encrypted will be encrypted; if the caller does not provide a payload, then the payload is the empty string.

The NK pattern is a variant of NN that provides authentication of the responder. The main difference is an additional message preceding the ellipses ($\leftarrow$ s) indicating that the responder's static public key is known to the initiator before the protocol begins. In its first action, the initiator computes the shared secret between this and its ephemeral secret (es) and uses it to encrypt the message payload. This has two effects: first, the initiator proves knowledge of the shared secret to the responder; and second, the responder authenticates itself by proving knowledge of the shared secret to the initiator. These properties are due to the sequence of actions induced by the pattern; if decryption fails, then this indicates that the sender does not know the correct shared secret. This works because each key derivation depends on all shared secrets computed in the protocol so far.

The NX pattern is similar except that the public key is *transmitted* to the initiator during the handshake, rather than out-of-band. For our purposes, the significant difference between NK and NX is that, in the former pattern, the initiator confirms knowledge of the shared secret *before* the responder consumes the message and produces its response. On the other hand, in the NX pattern the initiator can send an arbitrary element of the DH group as its ephemeral key and observe a valid response without demonstrating knowledge of its discrete logarithm. This leads to information leakage beyond what is learned by honest initiators (that is, for computationally bounded attackers). It is akin to providing the adversary with a functional DH oracle, which enables an attack against the GAP1 security of the interface; as we did in Theorem 4, one can exhibit a distinguisher that gets high advantage if the IDH problem is hard for the underlying group. (More on this attack in the full version [29].) To reason about this attack in our analysis, we require an abstraction for handshake patterns and the actions they induce.

Definition 10 (Patterns, actions, and tokenizers). A handshake pattern is a sequence of *message patterns* that specify the sequence of *tokens* processed when producing or consuming a message. A message pattern is a string that can be parsed by a *tokenizer*, which determines the set of valid *actions*. A tokenizer is a deterministic algorithm $\mathcal{T}(\textbf{bool}\, f, r, \textbf{str}\, \mathit{pat}) \mapsto \textbf{tup}\, \boldsymbol{t}, \textbf{str}\, \mathit{err}$. String pat is the message pattern, f indicates whether or not the host is producing a message, and r indicates whether the host is the initiator. The outputs are a tuple $\boldsymbol{t}$ comprised of the sequence of tokens to be processed and a string err indicating whether an error occurred. A valid action for $\mathcal{T}$ is a triple (f, r, pat) for which $\mathit{err} = \diamond$, where $(\boldsymbol{t}, \mathit{err}) = \mathcal{T}_f(r, \mathit{pat})$. We say that $\mathcal{T}$ has *action count* ℓ if $|\boldsymbol{t}| \le \ell$ for every valid action (f, r, pat).

A *token action* is a triple $(f, r, t) \in \{0,1\} \times \{0,1\} \times \{0,1\}^*$. We say that a tokenizer $\mathcal{T}$ *includes* a set of token actions $\mathcal{X}$ if for each $(f, r, t) \in \mathcal{X}$ the following is true: there exists a valid pattern pat for $\mathcal{T}$ such that $t = \boldsymbol{t}_i$ for some $1 \le i \le |\boldsymbol{t}|$ and $(\boldsymbol{t}, \mathit{err}) = \mathcal{T}_f(r, t)$. If this condition holds for no such token action, then $\mathcal{T}$ *excludes* $\mathcal{X}$. ♦

6.2 The Interface

The interface is specified as the composition of a tokenizer and the DH, AEAD, and hash primitives. Let $\mathbb{G} = \langle G \rangle$ be a v-encoded, additive group of order n, and fix integers $k, n', h, b, u \ge 0$ such that $v \notin \{u+8, h+u+8\}$. Let $\mathcal{AE}$ be an AEAD scheme with key-length k and nonce-length n'. Let $\mathit{cl} : \{0,1\}^b \to \mathbb{Z}_n \setminus \{0\}$, $\mathit{vr} : \{0,1\}^* \to \{0,1\}^u$, and $\mathcal{H} : \{0,1\}^* \to \{0,1\}^h$ be functions. Function $\mathcal{H}$ is a hash function that will serve multiple purposes, one of which is to derive symmetric keys using HKDF [20]. We will ignore the details of HKDF in this section and simply denote key derivation by a function $\mathcal{F} : (\{0,1\}^*)^3 \to (\{0,1\}^h)^3$ that maps an "information" string id, a "salt" X, and input key material Y to a triple of

```
Gen( )
1  $K \twoheadleftarrow \{0,1\}^b$
2  $s \leftarrow cl(K)$
3  ret $(\underline{sG}, \underline{s})$

4  dec struct {
5      str $P, E, S$ } msg
6  dec struct { str $id, psk$;
7      int $seq$; str $K, N$;
8      str $L, A$;
9      $Q, R \in \mathbb{G}$; $e \in \mathbb{Z}_n$ } st

Op($sk, ctx, op, in$)
10 dec st $hs$; msg $req$; bool $f, r$; str $u, pat, err$
11 $s \leftarrow \mathrm{Scal}(sk)$; $\underline{o, f, r, pat} \leftarrow op$; $\underline{hs, in} \leftarrow in$
12 if $o \neq$ noise $\vee\ hs.id \neq vr(ctx) \vee |hs.L| \neq h \vee$
13     $|hs.psk| \in \{u+8, h+u+8\}$ then ret $\perp$
14 if $f$ then   // outbound payload
15     $(resp, err) \twoheadleftarrow \mathrm{Write}(\&hs, s, r, pat, in)$
16     if $\neg err$ then ret $\underline{hs, resp, \diamond}$
17 else $\underline{req} \leftarrow in$   // inbound message
18     $(out, err) \twoheadleftarrow \mathrm{Read}(\&hs, s, r, pat, req)$
19     if $\neg err$ then ret $\underline{hs, out, \diamond}$
20 if $err$ ret $\underline{\diamond, \diamond, err}$
```

Fig. 7. Simple DL interface $\mathcal{N}$ for Noise. Let $\mathbb{G} = \langle G \rangle$ be a v-encoded, additive group of order n and let $h, b, u \geq 0$ be integers such that $v \notin \{u+8, h+u+8\}$. Let $cl : \{0,1\}^b \to \mathbb{Z}_n \setminus \{0\}$ and $vr : \{0,1\}^* \to \{0,1\}^u$ be functions. Procedures Write and Read are defined in the full version [29].

h-bit strings $\mathcal{F}(id, X, Y)$. We will model $\mathcal{F}$ as a random oracle in our analysis; in the full version [29] we address the implications of this modeling choice.

Figure 7 specifies our Noise interface $\mathcal{N}$ at a high level and defines structures **st** and **msg** for the handshake state and messages respectively. The key generator $\mathcal{N}$.Gen chooses a random, b-bit string K, sets $s \leftarrow cl(K)$, and returns $(\underline{sG}, \underline{s})$. (Thus, $\mathcal{N}$ is simple in the sense of Def. 8.) Function cl serves the same purpose as cl in our specification of EdDSA; it maps a bit string of a particular length to a suitable scalar s for use with the given group. The key operator $\mathcal{N}$.Op is defined in terms of two procedures:

- Read(&**st** hs, **int** s, **bool** r, **str** pat, **msg** req) $\mapsto$ **str** out, err. Called when consuming an inbound message. It takes as input the static key s and processes the action $(0, r, pat)$ on the message req and current handshake state hs. It outputs a payload out.
- Write(&**st** hs, **int** s, **bool** r, **str** pat, in) $\mapsto$ **msg** $resp$, **str** err. Called when producing an outbound message. It takes as input the static key s and processes the action $(1, r, pat)$ on the payload in and current handshake state hs. It outputs a message $resp$.

Read and Write are defined in terms of $\mathcal{T}$, $\mathcal{AE}$, $\mathcal{F}$, and $\mathcal{H}$. The operand encodes the current handshake state hs and the input in, and the operator op encodes an action (f, r, pat). If $f = 1$, then the host interprets in as a payload to send to its peer in its next handshake message; it calls Write and returns the updated state and outbound message. If $f = 0$, then the host interprets in as a message sent by the peer; it calls Read and returns the updated state and inbound payload.

Context-to-action binding. The context ctx is bound to the handshake state via a field $hs.id$, which should be equal to $vr(ctx)$ (7:12). Each call to $\mathcal{F}$ made by either Read or Write uses $hs.id$ as the label. In this way, interface $\mathcal{N}$ binds the string $hs.id = vr(ctx)$ to each key derivation, thereby binding the context to the *action* being performed. We call this *context-to-action* binding. This differs from Noise as it is, which uses an empty string as the information string for key derivation via HKDF (see [30, Section 4.3]). (Formally, the processing rules as they are specified are recovered by defining $vr(ctx) = \varepsilon$ for all ctx.) Noise binds the context to *initialization* of the handshake state (see [30, Section 5.3]), but action binding is required in our attack model in order to provide context separation when the game leaks its internal handshake state to the adversary. We will discuss the issue that arises in the next section.

In order to save space, we defer detailed explanation of the Noise parameters, as well as the complete specifications of Read and Write, to the full version of this paper [29]. These details are essential to understanding the proof of our main result (Theorem 7), but since the low-level details are cumbersome, we will focus the remainder on stating and interpreting our results.

6.3 Security

Interface $\mathcal{N}$ is GAP1 secure for any game $\mathcal{G}$ subject to the following restrictions. First, the tokenizer must exclude any write action involving DH on the static secret. (It may, however, read messages that depend on the static secret.) And second, each time $\mathcal{G}$ invokes $\mathcal{F}$ on an input (id, u, v) it must hold that $id = vr(\alpha)$, where α is the game context.

Fix Noise parameters $(\mathbb{G}, \mathcal{AE}, \mathcal{T}, \mathcal{H}, \mathcal{F}, cl, vr, k, n', h, b, u)$ and let $\mathcal{N}$ be the DL interface instantiated with these parameters as specified in Figure 7. Let $n = |\mathbb{G}|$ and let $\mathcal{X} = \{(1, 0, \mathsf{es}), (1, 0, \mathsf{ss}), (1, 1, \mathsf{se}), (1, 1, \mathsf{ss})\}$. Define $\psi : \{0,1\}^* \times (\{0,1\}^*)^3 \to \{0,1\}$ as the map $(ctx, (id, u, v)) \mapsto (vr(ctx) = id)$.

Theorem 7. *Suppose that n is prime. Let $\mathcal{G}$ be a regular game and suppose that $\mathcal{T}$ is $\mathcal{X}$-excluding and has action count ℓ. Let* **DDH** *be as defined in Figure 4. When $\mathcal{F}$ is modeled as a random oracle, there exists a regular,* **DDH**- *and* **Q**-*relative simulator $\mathcal{S}$ such that for all $t, q_G, q_I, q_R, c \geq 0$ there exists a $\hat{t}$-time CR-adversary $\mathcal{C}$ such that*

$$\mathbf{Adv}^{\mathrm{gap1}}_{\mathcal{N},\mathcal{G}}(\mathcal{S}, \boldsymbol{r}) \leq 2cq_I \mathbf{Adv}^{\mathrm{cr}}_{vr}(\mathcal{C}) + 2\ell q_I \mathbf{Adv}^{\text{int-ctxt}}_{\mathcal{AE}}(\hat{t}, 0, q_I)\,,$$

*where $\mathcal{G}$ is c-ro-bound and ψ-ro-regular; $\mathcal{AE}$, $\mathcal{T}$, $\mathcal{H}$, cl, and vr are 0-ro-bound; simulator $\mathcal{S}$ is $(O(t/(q_I{+}1)), q_I, \ell)$-resource, $\ell q_R q_I$-**DDH**-bound, and 2-**Q**-bound; $\boldsymbol{r} = (t, q_G, q_I, q_R)$; and $\hat{t} = O(t + q_R q_I)$.*

We will sketch the main ideas of the proof; refer to the full version for the details [29]. To simulate static DH computations on an input Y (either the peer's static or ephemeral key), the simulator $\mathcal{S}$ computes the set $\mathcal{V}$ of points incident to the adversary's RO queries. For each $Z \in \mathcal{V}$ it uses its DDH oracle to check if $(\log_G P)(\log_G Y) = \log_G Z$, where P is the host's static key. If so, then it uses Z to simulate the output of the interface. This is only possible in general for read actions, since these require the adversary to compute a ciphertext under the correct symmetric key, which can be obtained by querying the RO first. In fact, what we show is that, short of breaking the CR security of *vr* or INT-CTXT security of $\mathcal{AE}$, the only way to get a valid response from **Op** is to compute the inbound message as specified by the processing rules.

The need for context-to-action binding and the restriction of the game's RO queries arise in order to ensure there is no "subliminal channel" between the game and the adversary conveying information about the RO to the adversary beyond what it learns by making RO queries on its own. If the game provides the outputs of its RO queries to the adversary (e.g., by compromising the handshake state), then without action binding, these can be used by the adversary to compute ciphertexts without interacting with the RO. Hence, there is no way for the simulator to correctly respond given only knowledge of the adversary's RO queries. (Allowing the simulator to observe more RO queries than this—in particular, the game's—would make composition impossible.)

Finally, as we did in Section 5.2, we apply the GAP1 security of $\mathcal{N}$ and the composition theorem to the IDH problem for $\mathcal{N}$. We cannot reduce the CDH problem to it as we did in Corollary 1, since the simulator requires a **DDH** oracle. Of course, this is precisely what the GDH experiment provides. The following is obtained by applying Theorems 1 and 7. (We will not prove it, but the details are closely related to Corollary 1.)

Corollary 2. *Suppose that n is prime and that $\mathcal{T}$ is $\mathcal{X}$-excluding and has maximum action count ℓ. Let $r = |\mathrm{Rng}\, cl|$ and suppose that $r \mid 2^b$. Then for all $t, q_I, q_R \geq 0$ it holds that*

$$\mathbf{Adv}^{\mathrm{idh}}_{\mathbb{G},\mathcal{N}}(t, q_I, q_R) \leq n/r\mathbf{Adv}^{\mathrm{gdh}}_{\mathbb{G}}(O(t+\hat{q}), \ell q_R q_I) + 2\ell q_I \mathbf{Adv}^{\text{int-ctxt}}_{\mathcal{AE}}(\hat{t}, 0, q_I)\,,$$

where $\mathcal{F}$ is modeled as a random oracle; $\mathcal{AE}$, $\mathcal{T}$, $\mathcal{H}$, cl, and vr are 0-ro-bound; $\hat{q} = q_R + \ell(q_I + 1)$; and $\hat{t} = O(t + q_R q_I)$.

Remark 1. The use of the DDH oracle by the simulator in Theorem 7 is standard; it is used, for instance, to prove joint security of encryption and signing in the ROM [13]. In fact, the Noise spec calls for a group for which the GDH problem is hard; see [30, Section 4.1].

Acknowledgements

This work was made possible by NSF grant CNS-1816375. We thank the anonymous reviewers for their useful comments. We thank Trevor Perrin for his valuable feedback on our analysis of Noise.

References

1. Acar, T., Belenkiy, M., Bellare, M., Cash, D.: Cryptographic agility and its relation to circular encryption. In: Advances in Cryptology – EUROCRYPT 2010. pp. 403–422. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
2. Acar, T., Nguyen, L., Zaverucha, G.: A TPM Diffie-Hellman oracle. Cryptology ePrint Archive, Report 2013/667 (2013), `https://eprint.iacr.org/2013/667`

3. Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M.L., Stransky, C.: Comparing the usability of cryptographic APIs. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 154–171 (May 2017)
4. Barnes, R., Iyengar, S., Sullivan, N., Rescorla, E.: Delegated credentials for TLS. Internet-Draft draft-ietf-tls-subcerts-03, IETF Secretariat (February 2019), http://www.ietf.org/internet-drafts/draft-ietf-tls-subcerts-03.txt
5. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Proceedings of the 1st ACM Conference on Computer and Communications Security. pp. 62–73. CCS '93, ACM, New York, NY, USA (1993)
6. Bellare, M., Rogaway, P.: The security of triple encryption and a frameworkforcode-basedgame-playingproofs. In: Advances in Cryptology - EUROCRYPT 2006. pp. 409–426. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
7. Bellare, M., Tackmann, B.: The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In: Advances in Cryptology – CRYPTO 2016. pp. 247–276. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
8. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering **2**(2), 77–89 (Sep 2012)
9. Bhargavan, K., Boureanu, I., Fouque, P., Onete, C., Richard, B.: Content delivery over TLS: a cryptographic analysis of Keyless SSL. In: 2017 IEEE European Symposium on Security and Privacy (EuroS P). pp. 1–16 (April 2017). https://doi.org/10.1109/EuroSP.2017.52
10. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the rsa encryption standard PKCS #1. In: Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology. pp. 1–12. CRYPTO '98, Springer-Verlag, London, UK, UK (1998)
11. Brown, D.R.L., Gallant, R.P.: The static Diffie-Hellman problem. Cryptology ePrint Archive, Report 2004/306 (2004), https://eprint.iacr.org/2004/306
12. Camenisch, J., Chen, L., Drijvers, M., Lehmann, A., Novick, D., Urian, R.: One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 901–920 (May 2017)
13. Degabriele, J.P., Lehmann, A., Paterson, K.G., Smart, N.P., Strefler, M.: On the joint security of encryption and signature in EMV. In: Topics in Cryptology – CT-RSA 2012. pp. 116–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
14. Dowling, B., Paterson, K.G.: A cryptographic analysis of the wireguard protocol. In: Applied Cryptography and Network Security. pp. 3–21. Springer International Publishing, Cham (2018)
15. Gleeson, S., Zimman, C.: PKCS #11 cryptographic token interface base specification version 2.40. Online white paper (July 2015), http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html
16. Haber, S., Pinkas, B.: Securely combining public-key cryptosystems. In: Proceedings of the 8th ACM Conference on Computer and Communications Security. pp. 215–224. CCS '01, ACM, New York, NY, USA (2001)
17. Josefsson, S., Liusvaara, I.: Edwards-curve digital signature algorithm (EdDSA). RFC 8032, RFC Editor (January 2017)
18. Kelsey, J., Schneier, B., Wagner, D.: Protocol interactions and the chosen protocol attack. In: Security Protocols. pp. 91–104. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
19. Kobeissi, N., Bhargavan, K.: Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. Cryptology ePrint Archive, Report 2018/766 (2018), https://eprint.iacr.org/2018/766

20. Krawczyk, H., Eronen, P.: HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, RFC Editor (May 2010), http://www.rfc-editor.org/rfc/rfc5869.txt
21. Künnemann, R., Steel, G.: YubiSecure? Formal security analysis results for the Yubikey and YubiHSM. In: Security and Trust Management. pp. 257–272. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
22. Lim, C.H., Lee, P.J.: A key recovery attack on discrete log-based schemes using a prime order subgroup. In: Advances in Cryptology — CRYPTO '97. pp. 249–263. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
23. Lipp, B., Blanchet, B., Bhargavan, K.: A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol. Research Report RR-9269, Inria Paris (Apr 2019), https://hal.inria.fr/hal-02100345
24. Maurer, U., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Theory of Cryptography. pp. 21–39. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
25. Namprempre, C., Rogaway, P., Shrimpton, T.: Reconsidering generic composition. In: Advances in Cryptology – EUROCRYPT 2014. pp. 257–274. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
26. Okamoto, T., Pointcheval, D.: The gap-problems: A new class of problems for the security of cryptographic schemes. In: Public Key Cryptography. pp. 104–118. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
27. Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.C., Cappos, J., Zhuang, Y.: It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 296–305. ACSAC '14, ACM, New York, NY, USA (2014)
28. Oliveira, D.S., Lin, T., Rahman, M.S., Akefirad, R., Ellis, D., Perez, E., Bobhate, R., DeLong, L.A., Cappos, J., Brun, Y.: API blindspots: Why experienced developers write vulnerable code. In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018). pp. 315–328. USENIX Association, Baltimore, MD (2018)
29. Patton, C., Shrimtpon, T.: Security in the presence of key reuse: Context-separable interfaces and their applications. Cryptology ePrint Archive, Report 2019/519 (2019), https://eprint.iacr.org/2019/519
30. Perrin, T.: The Noise protocol framework. Online white paper (July 2018), https://noiseprotocol.org/noise.html
31. Pollard, J.M.: Kangaroos, monopoly and discrete logarithms. J. Cryptol. **13**(4), 437–447 (2000)
32. Rescorla, E.: The transport layer security (TLS) protocol version 1.3. RFC 8446, RFC Editor (August 2018)
33. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: Limitations of the indifferentiability framework. In: Advances in Cryptology – EUROCRYPT 2011. pp. 487–506. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
34. Rogaway, P., Stegers, T.: Authentication without elision: Partially specified protocols, associated data, and cryptographic models described by code. In: 2009 22nd IEEE Computer Security Foundations Symposium. pp. 26–39 (July 2009)
35. Shrimpton, T., Stam, M., Warinschi, B.: A modular treatment of cryptographic APIs: The symmetric-key case. In: Advances in Cryptology – CRYPTO 2016. pp. 277–307. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
36. Trusted Computing Group: TPM 2.0 library specification (September 2016), https://trustedcomputinggroup.org/resource/tpm-library-specification/