A preliminary version of this paper appears in the proceedings of PKC 2025. This is the full version.

Public-Algorithm Substitution Attacks: Subverting Hashing and Verification

Mihir Bellare[1] Doreen Riepel[2] Laura Shea[3]

March 14, 2025

Abstract

Algorithm Substitution Attacks (ASAs) have traditionally targeted secretly-keyed algorithms (for example, symmetric encryption or signing) with the goal of undetectably exfiltrating the underlying key. We initiate work in a new direction, namely ASAs on algorithms that are public, meaning contain no secret-key material. Examples are hash functions, and verification algorithms of signature schemes or non-interactive arguments. In what we call a PA-SA (Public-Algorithm Substitution Attack), the big-brother adversary replaces the public algorithm f with a subverted algorithm, while retaining a backdoor to the latter. Since there is no secret key to exfiltrate, one has to ask what a PA-SA aims to do. We answer this with definitions that consider big-brother's goal for the PA-SA to be three-fold: it desires utility (it can break an f-using scheme or application), undetectability (outsiders can't detect the substitution) and exclusivity (nobody other than big-brother can exploit the substitution). We start with a general setting in which f is arbitrary, formalizing strong definitions for the three goals, and then give a construction of a PA-SA that we prove meets them. We use this to derive, as applications, PA-SAs on hash functions, signature verification and verification of non-interactive arguments, exhibiting new and effective ways to subvert these. As a further application of the first two, we give a PA-SA on X.509 TLS certificates. Our constructions serve to help defenders and developers identify potential attacks by illustrating how they might be built.

[1] Department of Computer Science & Engineering, University of California San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. Email: mbellare@ucsd.edu. URL: http://cseweb.ucsd.edu/~mihir/. Supported in part by NSF grant CNS-2154272 and KACST.

[2] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany. Email: riepel@cispa.de. Work done while at UCSD, supported in part by KACST.

[3] Department of Computer Science & Engineering, University of California San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. Email: lmshea@ucsd.edu. Supported by NSF grants CNS-2048563, CNS-1513671 and CNS-2154272.

Contents

1 Introduction

The Snowden revelations lead researchers to consider different ways in which cryptography could be subverted. Algorithm Substitution Attacks (ASAs) [10, 69, 70] emerged as one answer. The formalism of Bellare, Paterson and Rogaway (BPR) [10] put forth the following template. There is a prescribed cryptographic algorithm A_k that accesses a secret key k. The adversary (called big-brother in this setting) substitutes A_k by subverted code $\tilde{\mathsf{A}}_{k,\tilde{k}}$ that continues to access k but now aims to undetectably exfiltrate it, for this purpose embedding and using another secret, symmetric key $\tilde{k}$ that it shares with big-brother.

Secret-algorithm substitution attacks. Conflating k and A_k, we can view the target algorithm A_k as secret. We will accordingly refer to ASAs such as the above as *secret-algorithm* substitution attacks (SA-SAs), to emphasize that the algorithm being substituted, namely A_k, is secret. Big-brother's goal in a SA-SA is to learn the secret algorithm.

ASAs have now been given or considered for many primitives including symmetric encryption [3, 9, 10, 27, 42], signature schemes [5, 21, 65] and beyond [14, 14, 20, 22, 39, 45, 56–58, 66]. *All of these continue to be SA-SAs*, meaning the target algorithm is secret. In the case of symmetric encryption, for example, the target algorithm is an encryption algorithm that is secret due to embedding the symmetric key; in the case of signatures, it is a signing algorithm that is secret due to embedding a secret signing key.

Public-algorithm substitution attacks. This paper initiates work in a new and different direction, namely what we introduce and call *public-algorithm* substitution attacks (PA-SAs). Here, the target algorithm is a public one; examples we will consider include hash functions, verification algorithms of signature schemes, or verification algorithms of non-interactive arguments (NIAs) like SNARGs or SNARKs. Our contributions can now be divided into three parts:

1. Definitions. With the target algorithm public, for example a public hash function, there is no secret to exfiltrate, so one has to ask what an ASA would want to do, and what properties big-brother would like it to have. We answer these questions with new definitions. Our setting for these is general, the target being an arbitrary public function f that the PA-SA substitutes with some $\widetilde{f}$, while retaining a related backdoor e. We require three properties, that we call utility (big-brother, through e, can make $\widetilde{f}$ behave differently from f), undetectability (outsiders can't detect the substitution) and exclusivity (nobody other than big-brother can exploit the substitution).

2. General PA-SA construction. Our definitions are strong, and the first question that emerges is whether a PA-SA meeting them is even possible. We show that it is, giving a general construction of a PA-SA on an arbitrary target public function f and proving that it meets our definitions.

3. Applications. To show that PA-SAs are a realistic threat, we use our general construction to give PA-SAs in three specific domains of practical interest. (1) The first is that f is a (any) public hash function, in which case our PA-SA allows the attacker to find structured collisions for $\widetilde{f}$. We obtain thence a forgery attack on X.509 TLS certificates and an attack on password-based authentication. (2) In our second application, f is the verification algorithm of a (any) NIA (Non-Interactive Argument), and the PA-SA allows violation of soundness, for *any* statement. NIAs are now seeing many uses, particularly in blockchains and cryptocurrencies, where our PA-SA highlights a new threat. (3) In our third application, f is the verification algorithm of a (any) signature scheme and the PA-SA allows signature forgery, for *any* verification key and *any* message.

PA-SAs are inspired by work on backdooring of hash functions [35] and machine-learning models [41] which our framework can capture as special cases that meet some of our requirements.

Overall, our work shows that PA-SAs are an effective attack vector of which developers and users should be aware. We now discuss the above in some more detail.

Our definitional framework. The target algorithm f, possibly together with some auxiliary information α, is sampled from a family F, as $(f, \alpha) \leftarrow\!\!\$\ \mathsf{F}$. Big-brother generates a substitution algorithm $\widetilde{f}$ together with an associated exploit algorithm (also called a backdoor) e, via $(\widetilde{f}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{F}}(f)$, where $\widetilde{\mathsf{F}}$, the PA-SA, is an algorithm of big-brother's own devising. Big-brother now arranges that a user's code implementing f is substituted with code implementing $\widetilde{f}$. (This substitution can take place in a variety of ways, for example via a malicious code update to a cryptographic library. How exactly it is done is outside our scope.) Applications that expected to use f are now (unknowingly) using $\widetilde{f}$ instead. We formalize three goals for big-brother, as follows:

Utility. Big-brother, through knowledge of the exploit algorithm e, wants to violate security of an $\widetilde{f}$-using application. Utility captures its ability to do so. While we might consider many forms of it, the one we choose and formalize is that knowledge of the exploit algorithm e allows big-brother to find a preimage x, under $\widetilde{f}$, of any output y of its choice. Importantly, x is not an arbitrary preimage, but one with a structure that big-brother can control; formally, utility ensures that $\mathsf{P}(x, u) = \mathsf{true}$ for some u of big-brother's choice, where P, called the constraint, is a predicate that parameterizes the definition. Different applications will make different choices of P, through which our single, parameterized definition will allow a wide range of different applications.

Undetectability. This was a core requirement for SA-SAs [10] and we accordingly require it for PA-SAs too. The formalization asks that a tester with blackbox access to either f or $\widetilde{f}$ should not be able to tell which of the two it is.

Exclusivity. We consider that big-brother does not want anyone other than itself to be able to make $\widetilde{f}$ behave pathologically. (Big-brother may be a government that, while itself having subversion capability, wants to ensure other governments do not.) Exclusivity captures this. The formalization considers an adversary that is given the descriptions of f and $\widetilde{f}$, and oracle access to the exploit function e, and asks that it still cannot come up with an input x at which f and $\widetilde{f}$ differ, except trivially, meaning through use of its oracle.

Exclusivity implies undetectability. We defined undetectability due to its central and historic presence in SA-SAs, but it turns out that exclusivity is even stronger. Namely, Theorem 3.1 says that exclusivity actually implies undetectability. Thus, for our PA-SAs, we will prove exclusivity and then conclude undetectability via Theorem 3.1.

Our general **SbvIt** construction. Is it possible to build a PA-SA $\widetilde{\mathsf{F}}$ on an arbitrary function family F that meets the three conditions above? We show, through construction, that the answer is "yes."

To expand on this, first note that one cannot hope to achieve this for all constraint predicates P. It is, for example, impossible for the predicate $\mathsf{P}(x, u)$ that returns true iff $x = u$; intuitively, one needs some "room" in x to exploit. We show how to build an ASA $\widetilde{\mathsf{F}}$ for any constraint predicate satisfying a certain condition, that we define and call *embeddability*. The class of predicates meeting this condition is large and includes in particular the above-discussed applications.

Our construction is a transform, called "Subvert-It" and denoted **SbvIt**. It takes (1) the target function family F (2) a signature scheme S and (3) an *embedding function* Emb, for the constraint predicate P, that is compatible with S, F, as we will define in Section 4. It returns a PA-SA $\widetilde{\mathsf{F}} = \mathbf{SbvIt}[\mathsf{F}, \mathsf{S}, \mathsf{Emb}]$ built from these three components. Proposition 4.1 establishes utility with respect to P, assuming correctness of S and Emb. Theorem 4.2 establishes exclusivity (and thus also undetectability) assuming strong unforgeability (suf-cma) of the signature scheme S.

The design of **SbvIt** extends ideas of [41], who used a signature scheme to backdoor a machine

learning model. Roughly, an input to $\widetilde{f}$ can be parsed as having two parts, the first an "instruction" to output something the instruction suggests, and the second a candidate signature of the first part. If the latter verifies under a verification key vk that $\widetilde{f}$ has in its code, then $\widetilde{f}$ executes the instruction, outputting whatever the latter decrees, else it returns the result of f on its input. The exploit e retains the signing key sk corresponding to vk so that e, and only it, can create valid signatures on instructions, and thus create inputs on which $\widetilde{f}$ behaves differently from f. Other entities are precluded from creating new "bad" inputs, even after having seen ones created by e, due to the suf-cma security of the signature scheme. See Section 4 for elaboration, including how making this work requires a condition that we capture in our definition of Emb being compatible with S, F.

PA-SAs on hash functions. Towards our first application, we start with some background. While algorithm substitution attacks on hash functions have not to our knowledge been discussed under that name, Fischlin, Janson and Mazaheri (FJM) [35] considered backdooring hash functions. Let H be the target set of hash functions from which, via $h \leftarrow\!\!\$\ \mathsf{H}$, one generates an honest hash function $h : \{0,1\}^* \to \{0,1\}^\ell$ that is assumed collision-resistant. A backdooring of H can be seen, like a PA-SA, as specified by an algorithm $\widetilde{\mathsf{H}}$ that takes an instance h of H and via $(\tilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}}(h)$ generates a substitution function $\tilde{h} : \{0,1\}^* \to \{0,1\}^\ell$ as well as a backdoor (or exploitation) function e. FJM [35] have two requirements that in our language represent utility and exclusivity. FJM-utility asks that knowledge of e allows big-brother to violate collision resistance (cr) of $\tilde{h}$, meaning find distinct x_1, x_2 such that $\tilde{h}(x_1) = \tilde{h}(x_2)$. FJM-exclusivity asks that an adversary given $\tilde{h}$ but not e cannot violate cr of $\tilde{h}$. FJM build a backdooring $\widetilde{\mathsf{H}}$ of H satisfying their two conditions. Other works also have backdoored hash functions satisfying these conditions: Albertini, Aumasson, Eichlseder, Mendel and Schläffer (AAEMS) [2] give such a backdoored version of $\mathsf{SHA1}$, and the VSH (Very Smooth Hash) algorithm of Contini, Lenstra and Steinfeld [23] achieves FJM-utility and FJM-exclusivity when viewed as a backdoored hash function.

FJM-utility is however limited; to subvert applications, big-brother needs, not just to be able to find some collision, but to create one in which the points embed structure, for example, as in our application, to have the form of a certificate. FJM-exclusivity is also limited, because it could be that after seeing a big-brother-produced collision, an adversary could violate cr. In Section 5, we give stronger definitions of utility and exclusivity for the particular case of PA-SAs on hash functions. We then show how to use our **SbvIt** transform to achieve them. We will illustrate for it two applications. One is subversion of password-based authentication, which we discuss at the end of Section 5, and the other, in Section 8, is X.509 certificate forgery in TLS.

To obtain the two exploits, we make particular choices of the constraint predicates that parameterize our definition. We then show that these predicates are embeddable, which allows us to apply our results about **SbvIt** to obtain the desired conclusions. In particular these applications rely crucially on our strong, predicate-parameterized definition of utility and are not possible under the prior definitions and for the prior constructions.

PA-SAs on non-interactive arguments. As a second example, we turn to non-interactive arguments (NIAs). The novel element here, and also in our application to signatures discussed below, is that the ASA substitutes the (public) *verification* algorithm rather than the (secret) proving or signing algorithm.

In the following, we use the notation $(v, p) \leftarrow\!\!\$\ \mathsf{NIA}$ to denote the generation of a verification algorithm, with the proving algorithm as auxiliary information. The CRS, if any, is hardcoded in these algorithms. Our PA-SA $\widetilde{\mathsf{NIA}}$ will replace v with a subverted function $\widetilde{v}$ for which big-brother holds an exploit function e that allows violation of soundness. Specifically, the definition of utility we make and achieve is very strong, asking that e allows the creation of an accepting proof for *any* statement which is not in the language corresponding to the NIA. Further, our exclusivity

definition asks that soundness is maintained relative to any party not having access to e, even if it has observed forged proofs, created under e, for statements of its choice. In Section 6 we show how to build such a PA-SA on *any* NIA by defining a suitable predicate and embedding function and then applying our **SbvIt** transform.

Prior work has considered building NIAs that resist CRS subversion [8, 37], namely retain soundness and/or ZK even in the presence of a malicious CRS. They however do not consider subversion of the verification algorithm, so our PA-SA would apply to their NIAs as well.

PA-SAs on signature schemes. Finally, we consider subversion of signature schemes. Prior work has been restricted to SA-SAs, which aim to exfiltrate the signing key by substituting either the key-generation [26] or signing [5, 65] algorithms. These SA-SA attacks and results, however, only work and apply when signing is randomized. In practical and standardized schemes, it often isn't. For example EdDSA [15], a NIST standard [52] that is the most prominent instantiation of Schnorr, makes signing deterministic. RFC 6979 [55] and NIST [52] both give deterministic modes for the otherwise randomized DSA and ECDSA schemes. In all these cases, the prior SA-SAs will not work. However, our PA-SA will. Indeed, our attacks and results apply to *any* signature scheme, including deterministic ones and those with unique signatures. This is made possible by subverting verification rather than key-generation or signing.

We now provide some more details on the attacker goals; a formal treatment is given in Section 7. Similar to NIAs, a signature scheme is formalized as an algorithm TS that, via $(v, kg, s) \leftarrow\!\!\$\ \mathsf{TS}$, generates a verifying algorithm and corresponding key-generation and signing algorithms, potentially hardcoding public parameters. A PA-SA on TS is an algorithm $\widetilde{\mathsf{TS}}$ that takes a target verification algorithm v and outputs a subverted verification algorithm $\widetilde{v}$ together with an exploit function e. Utility might ask that possession of exploit algorithm e allows creation of some forgery, meaning a message m and a valid signature for it under verification function $\widetilde{v}$ for some challenge vk^*. We ask for a stronger notion of utility, namely that e allows creation of a signature for *any target verification key* vk and *any target message* m, relative to $\widetilde{v}$. Our exclusivity definition is similarly strong, asking that unforgeability of signatures still holds even with access to an oracle for producing signatures using e.

Given a (any) signature scheme TS, we define a corresponding predicate and embedding function. Then we apply our **SbvIt** transform to obtain a PA-SA on TS that meets our above utility and exclusivity conditions. As a concrete application, we consider again the certificate forgery example. Instead of substituting the hash function, the exploitation function of our signature PA-SA allows big-brother to create certificates with validating signatures under a subverted verification algorithm. This is discussed in detail in Section 8.

Related work. "Subversion" is a recurring concern in cryptography. Attacks have been observed in a variety of settings, including parameter generation in Dual EC [16, 60], malicous code changes in Linux [33], and governmental exceptional access [1]. To situate our investigation of ASAs on public functions, it is useful to consider different categories in this area of subverted cryptography.

In a first category, code can be maliciously modified from its algorithmic specification. This has been studied as algorithm substitution attacks [10] and as kleptography [69–71]. Here, an adversary's goal is to both modify an algorithm such that it exfiltrates secret information, and to keep this modification undetected. Our attacks are similar in that they involve a code substitution step, and a second exploitation step. However, we don't exfiltrate secret information and instead target public algorithms. ASAs and kleptography have been studied for symmetric encryption [3, 9, 10, 27], KEMs [20, 45, 56], signatures [5, 65], and protocols [14, 14, 22, 39, 66]. Defenses have been studied from the perspective of preventing exfiltration [31, 51] or other "subversion-resistant" notions [4, 13, 57, 58]. These include more fine-grained online/offline detectability notions and other

modes of computation.

Another category of subversion work considers an authority that knows user secret keys. Anamorphic cryptography [7, 48, 54] and earlier work on subliminal channels [44, 61, 62] have proposed defenses to this type of subversion.

A final category, somewhat more related to ASAs, is maliciously designed algorithms or parameters. These have been studied for PRGs [28, 30], NIZKs [8], PKE [6], and hash functions [2, 35]. Unlike an ASA, algorithms are assumed to be implemented honestly, and code can be inspected. Nonetheless, some of the goals and techniques are similar.

Goldwasser, Kim, Vaikuntanathan and Zamir (GKVZ) give a way to insert undetectable backdoors in a machine learning model [41] using a strongly unforgeable signature scheme. Their classifier modifies the output of the correct classifier when a signature is correctly parsed and verified. The non-replicability condition of GKVZ offers an oracle providing backdoored model inputs, which is similar to what our exclusivity game captures. We note that the predicates and embeddings we additionally formalize in Section 4 generalize the parsing of GKVZ, and that our techniques for "general public functions" seem to be quite applicable beyond cryptographic functions.

As a real-world motivation to study ASAs, the `xz` backdoor was discovered on March 29, 2024 [36]. Current understanding of its cryptographic portion [67] shows interesting similarities with our PA-SAs, such as the embedding of a signature and attacker-chosen data in a certificate which triggers alternate execution during certificate validation. The discovery of the `xz` backdoor shows that ASAs targeting high levels of utility are a realistic possibility, and motivates research, such as ours, on this topic.

2 Preliminaries

NOTATION AND TERMINOLOGY. By ε we denote the empty string. By $|Z|$ we denote the length of a string Z. By $x \,\|\, y$ we denote the concatenation of strings x, y. If Z is a string, we let $Z[a..b]$ be the substring of Z between indices a and b, inclusive, or ε if $b < a$. If S is a finite set, then $|S|$ denotes it size. We say that a set S is *length-closed* if, for any $x \in S$ it is the case that $\{0,1\}^{|x|} \subseteq S$. (This will be a requirement for certain spaces.)

If $\mathcal{X}$ is a finite set, we let $x \leftarrow\!\!\$\, \mathcal{X}$ denote picking an element of $\mathcal{X}$ uniformly at random and assigning it to x. If A is an algorithm, we let $y \leftarrow A[\mathrm{O}_1, \ldots](x_1, \ldots; r)$ denote running A on inputs $x_1, \ldots$ and coins r with oracle access to $\mathrm{O}_1, \ldots$, and assigning the output to y. We let $y \leftarrow\!\!\$\, A[\mathrm{O}_1, \ldots](x_1, \ldots)$ be the result of picking r at random and computing $y \leftarrow A[\mathrm{O}_1, \ldots](x_1, \ldots; r)$. We let $\mathrm{OUT}(A[\mathrm{O}_1, \ldots](x_1, \ldots))$ denote the set of all possible outputs of A when invoked with inputs $x_1, \ldots$ and oracles $\mathrm{O}_1, \ldots$. Algorithms are randomized unless otherwise indicated. Running time is worst case, which for an algorithm with access to oracles means across all possible replies from the oracles.

We will often speak of an algorithm that has, as input or output, other algorithms, conflating here the algorithms with their descriptions. For this purpose, we can assume some programming or description language has been fixed. We also conflate deterministic algorithms with functions.

An adversary is an algorithm. We use $\perp$ (bot) as a special symbol to denote rejection, and it is assumed to not be in $\{0,1\}^*$. The image of a function $f : \mathcal{D} \to \mathcal{R}$ is the set $\mathrm{Im}(f) = \{f(x) : x \in \mathcal{D}\} \subseteq \mathcal{R}$. We may interchangeably refer to the boolean false and integer 0, or to the boolean true and integer 1.

GAMES. We use the code-based game-playing framework of BR [11]. A game G starts with an optional INIT procedure, followed by a non-negative number of additional procedures called oracles,

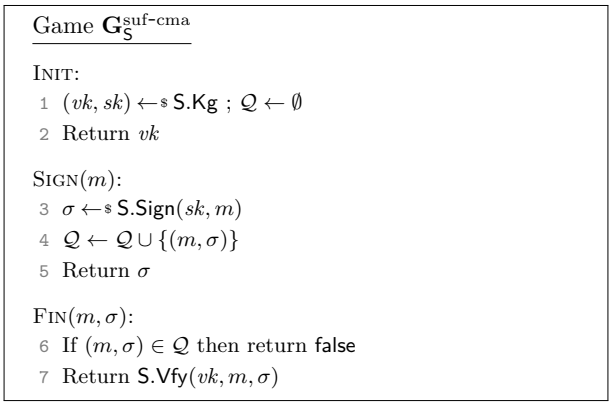

Figure 1: Strong unforgeability for a signature scheme S.

and ends with a FIN procedure. Execution of adversary A with game G begins by running INIT (if present) to produce $\mathsf{input} \leftarrow\!\!\$\ \text{INIT}$. A is then given input and is run with query access to the game oracles. When A terminates with some output, execution of game G ends by returning FIN(output). By $\Pr[\text{G}(A)]$ we denote the probability that the execution of game G with adversary A results in FIN(output) being the boolean true.

Different games may have procedures (oracles) with the same names. If we need to disambiguate, we may write G.O to refer to oracle O of game G. In games, integer variables, set variables, boolean variables and string variables are assumed initialized, respectively, to 0, the empty set $\emptyset$, the boolean false and $\perp$. Tables are initialized with all entries being $\perp$. Games may occasionally Require: some condition, which means that all adversaries must obey this condition. This is used to rule out trivial wins.

UNFORGEABILITY OF SIGNATURES. A signature scheme S specifies algorithms $\mathsf{S.Kg}, \mathsf{S.Sign}, \mathsf{S.Vfy}$, key spaces $\mathsf{S.VK}, \mathsf{S.SK}$, and signature length S.sl. Key generation S.Kg produces a verification key $vk \in \mathsf{S.VK}$ and signing key $sk \in \mathsf{S.SK}$ via $(vk, sk) \leftarrow\!\!\$\ \mathsf{S.Kg}$. Signing takes as input a signing key $sk \in \mathsf{S.SK}$ and message $m \in \{0,1\}^*$ to return a signature $\sigma \in \{0,1\}^{\mathsf{S.sl}}$ via $\sigma \leftarrow\!\!\$\ \mathsf{S.Sign}(sk, m)$, where $\mathsf{S.sl} \in \mathbb{N}$ is a constant signature length. Deterministic algorithm S.Vfy takes as input a verification key $vk \in \mathsf{S.VK}$, message $m \in \{0,1\}^*$, and signature $\sigma \in \{0,1\}^{\mathsf{S.sl}}$ to return a bit d via $d \leftarrow \mathsf{S.Vfy}(vk, m, \sigma)$.

Correctness of scheme S asks that for all $(vk, sk) \in \text{OUT}(\mathsf{S.Kg})$, for all $m \in \{0,1\}^*$, it holds that $\mathsf{S.Vfy}(vk, m, \mathsf{S.Sign}(sk, m)) = 1$.

The security notion that we will make use of is strong unforgeability. This is captured by game $\mathbf{G}^{\text{suf-cma}}_{\mathsf{S}}$ of Figure 1. If A is an adversary, we let $\mathbf{Adv}^{\text{suf-cma}}_{\mathsf{S}}(A) = \Pr\left[\mathbf{G}^{\text{suf-cma}}_{\mathsf{S}}(A)\right]$ be its suf-cma advantage. Strongly unforgeable signatures have been constructed based on bilinear CDH [18], strong RSA [25, 38], and generally from one-way functions [40, Section 6.5].

3 Public-Algorithm Substitution Attacks

We begin with new definitions for algorithm substitution attacks on arbitrary public algorithms. In Section 4 we will provide a construction satisfying our notions. In the remainder of the paper we apply this to more specific settings. As our first task, we introduce a generic PA-SA syntax.

SYNTAX. A family is an algorithm F that, via $(f, \alpha) \leftarrow\!\!\$\ \mathsf{F}$, creates an algorithm $f : \{0,1\}^* \rightarrow \{0,1\}^{\mathsf{F.ol}}$

together with auxiliary information α. Here $\mathsf{F.ol} \in \mathbb{N}$ is an output length associated to F and f is the "public algorithm" that is the target of subversion.

A generic PA-SA aims to substitute f with $\widetilde{f}$ which has certain malicious behavior on particular inputs, while ensuring that the ability to find such inputs is exclusive to the attacker. Formally, the PA-SA is an algorithm $\widetilde{\mathsf{F}}$ which takes f as input to return $(\widetilde{f}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{F}}(f)$, where $\widetilde{f}$ is called the *substitution algorithm* and e is the *exploitation algorithm*. Before elaborating on these components, let us clarify the PA-SA model.

PA-SA MODEL. An attacker who mounts a public-algorithm substitution attack proceeds in two steps, substitution and exploitation. First, they generate algorithms $(\widetilde{f}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{F}}(f)$ and replace a user's implementation of f with one of $\widetilde{f}$. The exploitation algorithm e remains secret and is retained by the attacker. Second, in the exploitation step, the attacker may use e to create an input x for $\widetilde{f}$ under which the latter behaves in some way favorable to the attacker. We focus on a particular and powerful form of this behavior that we will show allows subversion in many applications. Namely, the attacker has a target output point y and aims to create a preimage of it, more specifically an x such that $\widetilde{f}(x) = y$, where x is not arbitrary but instead has a form or content determined by the attacker. When this is done, the user, who has $\widetilde{f}$ on their device and receives x, computes $\widetilde{f}(x) = y$. The interface of this exploitation algorithm is broad, applying to settings where (1) it is reasonable that the attacker chooses inputs x to send to the user, and (2) it is useful to an attacker to find preimages of target points — where the target point could even be 1 or true.

What is the auxiliary information α? At a high level, this includes information that is associated to f and available in the world, but is not necessarily needed by the attacker. In particular, α may aid honest parties in detecting whether f has been substituted by $\widetilde{f}$. It is not always relevant or needed; we will see, for example, that α is empty for hash functions (where only one public algorithm is generated) but it will be non-empty in our applications to non-interactive arguments and signatures (where multiple algorithms are co-generated).

CONSTRAINT PREDICATES. As above, the attacker aims, using the exploit algorithm, to create some structured x satisfying $\widetilde{f}(x) = y$. The desired structure depends on the application. Constraint predicates are how we formalize and capture this. Our definitions will be parameterized by a predicate, and we will then aim to build PA-SAs for as large as possible a class of predicates.

Formally, a constraint predicate is a function $\mathsf{P} : \{0,1\}^* \times \{0,1\}^* \to \{\mathsf{true}, \mathsf{false}\}$. The second input is called the constraint-parameter. A variety of predicates may be desirable. Predicate $\mathsf{P}(x, u)$ could capture whether x is a valid $\mathsf{X.509}$ certificate containing information u; this is considered in more detail in Section 8. A predicate could capture whether x can be parsed as human-readable text or otherwise does not "look suspicious." Increasingly useful predicates will come with implementation challenges beyond mounting a PA-SA, but the ones we will describe in Section 4 are already potent.

UTILITY. The *utility* of $\widetilde{\mathsf{F}}$ is measured with respect to a constraint predicate P. We say that $\widetilde{\mathsf{F}}$ achieves P-utility, if for every f output by F, every constraint-parameter $u \in \{0,1\}^*$ and every $y \in \{0,1\}^{\mathsf{F.ol}}$, if $(\widetilde{f}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{F}}(f)$ and if $x \leftarrow\!\!\$\ e(u, y)$, then we have (1) $\widetilde{f}(x) = y$ and (2) $\mathsf{P}(x, u) = 1$. In other words, e allows one to compute a preimage of any target output, where the preimage also satisfies the constraint predicate. In the following section we discuss predicates in more detail. At a high level, this utility definition relative to a predicate allows one to specify fine-grained notions of successful attacks, beyond only finding *some* preimage.

EXCLUSIVITY. The exploit algorithm allows its possessor to create inputs on which $\widetilde{f}$ and f might differ. *Exclusivity* requires that the exploit algorithm is necessary for this. The formalization, via the exclusivity game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}$ of Figure 2, is strong. Via oracle GETPMG, the adversary A can choose

Game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}$	Game $\mathbf{G}^{\mathrm{det}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}$
Init:	Init:
1 $(f,\alpha) \leftarrow\!\!\$\ \mathsf{F}$; $(\widetilde{f},e) \leftarrow\!\!\$\ \widetilde{\mathsf{F}}(f)$	1 $(f,\alpha) \leftarrow\!\!\$\ \mathsf{F}$; $(\widetilde{f},e) \leftarrow\!\!\$\ \widetilde{\mathsf{F}}(f)$
2 $\mathcal{X} \leftarrow \emptyset$	2 $b \leftarrow\!\!\$\ \{0,1\}$
3 Return $(f,\widetilde{f},\alpha)$	3 Return ε
GetPmg(u,y):	Eval(x):
4 $x \leftarrow\!\!\$\ e(u,y)$; $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$	4 $y_0 \leftarrow f(x)$; $y_1 \leftarrow \widetilde{f}(x)$
5 Return x	5 Return y_b
Fin(x^*):	Fin(b'):
6 Return ($x^* \notin \mathcal{X}$ and $f(x^*) \neq \widetilde{f}(x^*)$)	6 Return $(b = b')$

Figure 2: Exclusivity (left) and detectability (right) of a PA-SA $\widetilde{\mathsf{F}}$ on f. While detectability is blackbox, exclusivity returns the functions $(f,\widetilde{f})$ and auxiliary information α to the adversary.

---

inputs on which to see outputs of the exploit algorithm. To win, it must produce an input x^* which was not generated by GetPmg, yet on which $\widetilde{f}$ and f differ. We let $\mathbf{Adv}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A) = \Pr\left[\mathbf{G}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A)\right]$ be the exclusivity advantage of A.

We will see exclusivity applied to more specific settings in later sections, but for now, we remark that a PA-SA is only useful if the substituted algorithm $\widetilde{f}$ differs from f on some inputs (else f could be attacked directly). Exclusivity asks that is hard for anyone other than the PA-SA attacker to find these inputs which produce different behavior.

Undetectability. Undetectability is a core requirement in ASAs [10]. Here we formulate it in our setting of PA-SAs. The game is on the right side of Figure 2. If A is an adversary, we let $\mathbf{Adv}^{\mathrm{det}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A) = 2\cdot\Pr\left[\mathbf{G}^{\mathrm{det}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A)\right] - 1$ be its (un)detectability advantage. As with prior formulations, the definition is of blackbox undetectability, meaning the adversary is not given the functions $f,\widetilde{f}$ themselves, but only access to an oracle for them, the definition asking that such access not allow the adversary to distinguish the two.

We define undetectability since, as indicated above, it is the norm for ASAs, but in fact the following says that our new exclusivity notion implies undetectability. This allows us to focus, moving forward, on the stronger exclusivity notion.

Theorem 3.1 *Let F be a family and $\widetilde{\mathsf{F}}$ a PA-SA on F relative to a predicate P. Given an adversary A against the undetectability of $\widetilde{\mathsf{F}}$ we can build an adversary A' such that*

$$\mathbf{Adv}^{\mathrm{det}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A) \leq 2 \cdot \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A') \ . \tag{1}$$

A' makes no GetPmg queries. The running time of A' is close to that of A.

Proof of Theorem 3.1: Consider game G_0 of Figure 3 which is exactly the detectability game, except that it additionally checks whether y_0 equals y_1 for queries to Eval and sets flag bad if they are not the same. The output, however, is not modified, so

$$\mathbf{Adv}^{\mathrm{det}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A) = 2 \cdot \Pr\left[\,\mathrm{G}_0(A)\,\right] - 1 \ .$$

We now turn to G_1. Games $\mathrm{G}_0, \mathrm{G}_1$ are identical-until-bad, so by the Fundamental Lemma of Game

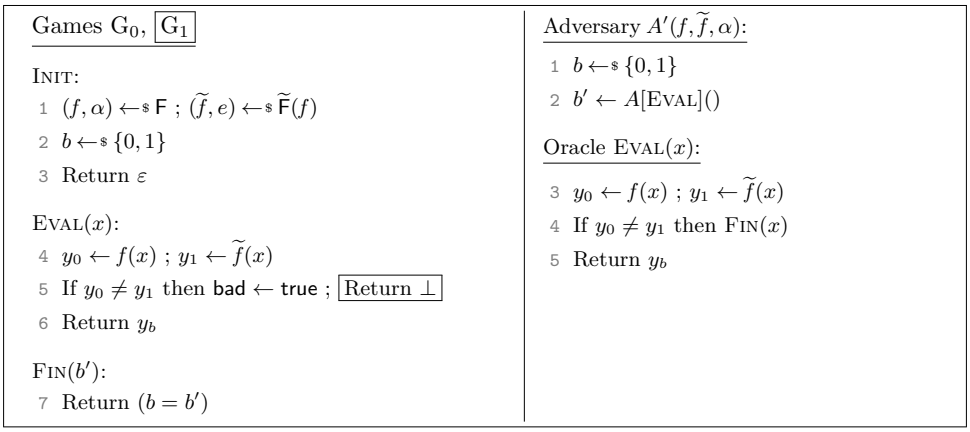

Figure 3: Games G_0, G_1 (left) and adversary A' (right) for the proof of Theorem 3.1. G_1 contains the boxed code and G_0 does not.

Playing [11] we have

$$\begin{aligned}\Pr[\,G_0(A)\,] &= \Pr[\,G_1(A)\,] + (\Pr[\,G_0(A)\,] - \Pr[\,G_1(A)\,]) \\ &\leq \Pr[\,G_1(A)\,] + \Pr[\,G_1(A) \text{ sets } \mathsf{bad}\,]\ .\end{aligned}$$

We construct adversary A' on the right side of Figure 3. A' gets as input algorithms f and $\widetilde{f}$, and can simulate the detectability game in a straightforward way by picking its own challenge bit. Since A' does not ask any GETPMG queries, then if bad is set in G_1, A' has found a winning output in the exclusivity game. We have

$$\Pr[\,G_1(A) \text{ sets } \mathsf{bad}\,] \leq \mathbf{Adv}^{\text{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A')\ .$$

Finally note that $\Pr[\,G_1(A)\,] = 1/2$ since the outputs of EVAL are independent of the challenge bit. Collecting the probabilities proves Eq. (1). ∎

4 PA-SA construction

EMBEDDINGS. We first introduce message embeddings as a way to realize constraint predicates in our PA-SA construction. A message embedding for constraint predicate P is a function $\mathsf{Emb} : \mathsf{Emb.ES} \times \{0,1\}^* \to \{0,1\}^*$, where $\mathsf{Emb.ES}$ is a set (the "embedding space") that must be specified and depends on the intended predicate. There is also an inverse $\mathsf{EmbInv} : \{0,1\}^* \to (\mathsf{Emb.ES} \times \{0,1\}^*) \cup \{\perp\}$ such that (1) for all $(z,u) \in \mathsf{Emb.ES} \times \{0,1\}^*$, if $x \leftarrow \mathsf{Emb}(z,u)$ then $\mathsf{P}(x,u) = 1$ and $\mathsf{EmbInv}(x) = (z,u)$, and (2) $\mathsf{EmbInv}(x) = \perp$ for all $x \notin \text{Im}(\mathsf{Emb})$. We say that Emb is a *correct embedding function* for predicate P if these two properties are satisfied. In other words, Emb is a bijection from $\mathsf{Emb.ES} \times \{0,1\}^*$ to $\text{Im}(\mathsf{Emb})$ with inverse EmbInv.

Two illustrative examples are prefix and suffix embeddings. Suppose one wants to find a preimage x of output y, with the constraint that x begins with prefix u, or ends with suffix u. For a prefix embedding, let $n \in \mathbb{N}$ and $\mathsf{Emb}^n_{\mathsf{pfx}}.\mathsf{ES} = \{0,1\}^n$. Then the predicate $\mathsf{P}^n_{\mathsf{pfx}}$ and embedding function $\mathsf{Emb}^n_{\mathsf{pfx}}$ are given on the left side of Figure 4. Similarly, for suffixes, let $\mathsf{Emb}^n_{\mathsf{sfx}}.\mathsf{ES} = \{0,1\}^n$, with the predicate $\mathsf{P}^n_{\mathsf{sfx}}$ and embedding function $\mathsf{Emb}^n_{\mathsf{sfx}}$ on the right side of Figure 4. The choice of n here is important: it will matter for the feasibility of designing an effective PA-SA in some settings.

$\underline{\mathsf{P}^n_{\mathsf{pfx}}(x,u)}$:
1 If $(|x| \neq |u|+n)$ then return false
2 Return $(x[1..|u|] = u)$

$\underline{\mathsf{Emb}^n_{\mathsf{pfx}}(z,u)}$:
3 Return $u \,\|\, z$

$\underline{\mathsf{EmbInv}^n_{\mathsf{pfx}}(x)}$:
4 If $(|x| < n)$ then return $\perp$
5 $z \leftarrow x[(|x|-n+1)..|x|]$
6 $u \leftarrow x[1..(|x|-n)]$
7 Return (z,u)

$\underline{\mathsf{P}^n_{\mathsf{sfx}}(x,u)}$:
1 If $(|x| \neq |u|+n)$ then return false
2 Return $(x[(|x|-|u|+1)..|x|] = u)$

$\underline{\mathsf{Emb}^n_{\mathsf{sfx}}(z,u)}$:
3 Return $z \,\|\, u$

$\underline{\mathsf{EmbInv}^n_{\mathsf{sfx}}(x)}$:
4 If $(|x| < n)$ then return $\perp$
5 $z \leftarrow x[1..n]$
6 $u \leftarrow x[(n+1)..|x|]$
7 Return (z,u)

Figure 4: Practical examples of predicates and embedding functions: a prefix embedding (left) and suffix embedding (right). The embedding space is $\mathsf{Emb}^n_{\mathsf{pfx}}.\mathsf{ES} = \mathsf{Emb}^n_{\mathsf{sfx}}.\mathsf{ES} = \{0,1\}^n$ for a fixed n.

$\underline{\widetilde{\mathsf{F}}_0(f)}$:
1 $bd \leftarrow\!\!\$\ \{0,1\}^k$; $t \leftarrow g(bd)$
2 Define as below:
$\widetilde{f}_t : \{0,1\}^* \to \{0,1\}^{\mathsf{F.ol}}$
$e_{bd} : \{0,1\}^* \times \{0,1\}^{\mathsf{F.ol}} \to \{0,1\}^*$
3 Return $(\widetilde{f}_t, e_{bd})$

$\underline{\widetilde{f}_t(x)}$:
4 If $(|x| = k + \mathsf{F.ol})$ then
5 $bd' \leftarrow x[1..k]$
6 If $(g(bd') = t)$ then return $x[(k+1)..|x|]$
7 Else return $f(x)$

$\underline{e_{bd}(u,y)}$:
8 // In this warmup, u is ignored
9 Require: $y \in \{0,1\}^{\mathsf{F.ol}}$
10 $x \leftarrow bd \,\|\, y$
11 Return x

$\underline{\widetilde{\mathsf{F}}(f)}$:
1 $(vk, sk) \leftarrow\!\!\$\ \mathsf{S.Kg}$
2 Define as below:
$\widetilde{f}_{vk} : \{0,1\}^* \to \{0,1\}^{\mathsf{F.ol}}$
$e_{sk} : \{0,1\}^* \times \{0,1\}^{\mathsf{F.ol}} \to \{0,1\}^*$
3 Return $(\widetilde{f}_{vk}, e_{sk})$

$\underline{\widetilde{f}_{vk}(x)}$:
4 $w \leftarrow \mathsf{EmbInv}(x)$
5 If $(w = \perp)$ then return $f(x)$
6 $(y \,\|\, \sigma, u) \leftarrow w$
7 If $\mathsf{S.Vfy}(vk, (y,u), \sigma)$ then return y
8 Else return $f(x)$

$\underline{e_{sk}(u,y)}$:
9 Require: $y \in \{0,1\}^{\mathsf{F.ol}}$
10 $\sigma \leftarrow\!\!\$\ \mathsf{S.Sign}(sk, (y,u))$
11 $x \leftarrow \mathsf{Emb}(y \,\|\, \sigma, u)$
12 Return x

Figure 5: **Left:** Warmup construction of a PA-SA $\widetilde{\mathsf{F}}_0$ using a one-way function g. **Right:** Construction of a PA-SA $\widetilde{\mathsf{F}}$, relative to predicate P with associated embedding function Emb, using a signature scheme S. In this figure, subscripts denote hardcoded information (t, bd, vk, sk). In the future these may be omitted from subscripts for brevity.

Warmup construction. We begin with a basic construction of a PA-SA. This is essentially a generalization of the construction of FJM [35, Section 7.1] which targets backdoored hash functions. More specifically, we build a PA-SA algorithm $\widetilde{\mathsf{F}}_0$ for an algorithm instance f using a one-way function $g : \{0,1\}^k \to \{0,1\}^\ell$. $\widetilde{\mathsf{F}}_0$ is specified on the left of Figure 5. To an entity with e_{bd}, finding a preimage of target y is simple: $bd \,\|\, y$ is a preimage because the trigger $(g(bd') = t)$ passes in $\widetilde{f}_t$, which then returns y. As we now explain, however, this construction falls short of our requirements.

$\widetilde{\mathsf{F}}_0$ satisfies P-utility for the predicate $\mathsf{P}(x,u) = 1$ for all x, u. This is quite weak, saying

effectively that, while big-brother can find a preimage x of a given y, it has no control over x and no ability to structure x in a favorable way. Our construction instead will provide P-utility for a large class of predicates P.

In terms of exclusivity, t is public but finding bd remains hard to find assuming g is one-way. However, this does not hold if a preimage produced by e_{bd} is observed. Thus our notion of exclusivity is not satisfied, but blackbox undetectability is. We omit a formal analysis as we will next consider a construction which does meet all of our notions.

OUR CONSTRUCTION. We now give a construction meeting all our requirements. Let S be an suf-cma signature scheme and let F be a family of algorithms. We say that message embedding function $\mathsf{Emb} : \mathsf{Emb.ES} \times \{0,1\}^* \to \{0,1\}^*$ is *compatible* with S, F if $\mathsf{Emb.ES} = \{0,1\}^{\mathsf{F.ol}+\mathsf{S.sl}}$. That is, the embedding information consists of an output of an algorithm in F and a signature. Our transform **SbvIt** associates to S, F, and an Emb compatible to S, F a PA-SA algorithm $\widetilde{\mathsf{F}} = \mathbf{SbvIt}[\mathsf{F}, \mathsf{S}, \mathsf{Emb}]$ which is defined on the right side of Figure 5.

Note that we assume a correct embedding function Emb for predicate P. These have been given for common predicates in Figure 4, and can be given for a large class of predicates including ones sufficient for the applications we will give in later sections. But it may not be the case that every predicate has a correct embedding function, or that every embedding function is compatible with S, F.

Our use of an suf-cma signature scheme follows [41], but predicates and embeddings are new and our construction is more general than theirs. The inclusion of this signature verification in $\widetilde{f}_{vk}$ (line 7 on the right side of Figure 5) also raises a question of timing-based detectability. This has always been a question and possibility with ASAs, and has been outside the formal models. For example, the SA-SA of [9] uses rejection sampling, so subverted encryption is slower than regular. Timing-based detection remains possible in our construction but there are mitigating elements in the applications we consider. Hash functions (Section 5) are often used within more time-consuming applications (such as certificate validation as in Section 8) so the overall change in time could be small. Similarly, for signatures (Section 7), the overhead and original cost are close.

In the remainder of this section, we show that $\widetilde{\mathsf{F}}$ produced by transform $\mathbf{SbvIt}[\mathsf{F}, \mathsf{S}, \mathsf{Emb}]$ achieves P-utility as long as S, Emb are correct (Proposition 4.1) and achieves exclusivity as long as S is suf-cma (Theorem 4.2). The bottom line of this result is that effective PA-SAs *are* possible to construct from standard building blocks, for useful constraints.

Proposition 4.1 *Let S be a signature scheme, F a family, and Emb an embedding function for predicate P which is compatible with S, F. Let $\widetilde{\mathsf{F}} = \mathbf{SbvIt}[\mathsf{F}, \mathsf{S}, \mathsf{Emb}]$. If S is a correct signature scheme and Emb is a correct embedding function for predicate P, then $\widetilde{\mathsf{F}}$ achieves P-utility.*

Proof of Proposition 4.1: Consider any $(f, \alpha) \in \mathrm{OUT}(\mathsf{F})$, $(\widetilde{f}_{vk}, e_{sk}) \in \mathrm{OUT}(\widetilde{\mathsf{F}}(f))$, $u \in \{0,1\}^*$, and $y \in \{0,1\}^{\mathsf{F.ol}}$. The algorithm e_{sk}, on inputs u and y, returns $x \leftarrow \mathsf{Emb}((y \,\|\, \sigma), u)$ where $\sigma \leftarrow\!\!\$\ \mathsf{S.Sign}(sk, (y, u))$. Property (1) of utility requires that $\widetilde{f}(x) = y$. Let us consider $\widetilde{f}_{vk}(x)$ of our construction. On lines 4,5, since $x \in \mathrm{Im}(\mathsf{Emb})$, $w \neq \bot$. If $\mathsf{Emb}, \mathsf{EmbInv}$ satisfy our notion of a correct embedding, w is recovered as $((y \,\|\, \sigma), u)$. That is, $\mathsf{EmbInv}(\mathsf{Emb}((y \,\|\, \sigma), u)) = ((y \,\|\, \sigma), u)$. Next, the signature verification on line 7 runs $\mathsf{S.Vfy}(vk, (y, u), \sigma)$ where $\sigma \leftarrow\!\!\$\ \mathsf{S.Sign}(sk, (y, u))$. This passes as long as S is a correct signature scheme, and $\widetilde{f}_{vk}(x)$ thus returns y on line 7.

Property (2) of utility asks that $\mathsf{P}(x, u) = 1$. This is proven by line 11, where $x \leftarrow \mathsf{Emb}((y \,\|\, \sigma), u)$. Correctness of the embedding function Emb for predicate P implies that $\mathsf{P}(x, u) = 1$. ∎

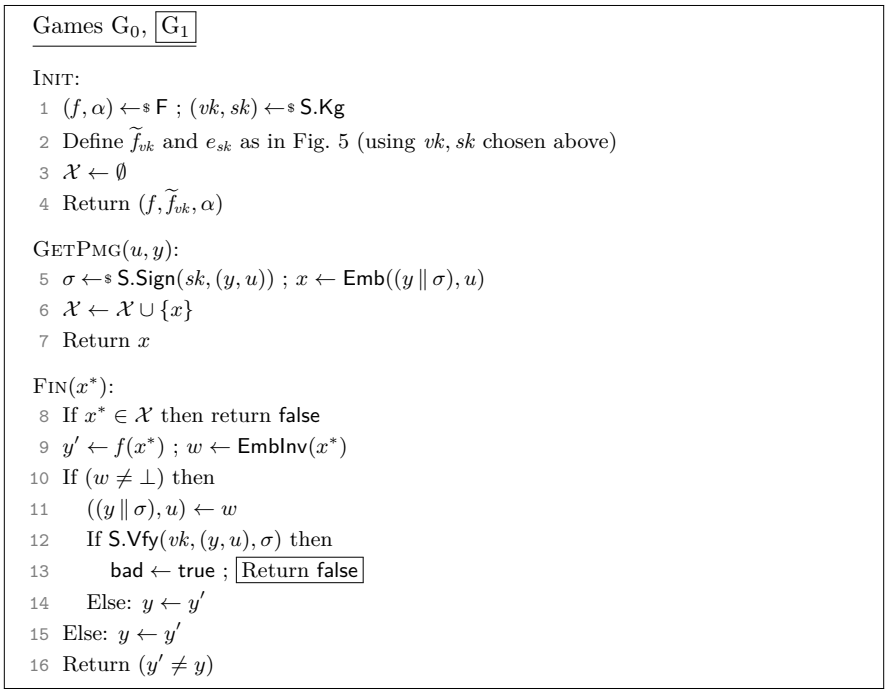

Figure 6: Games $\mathrm{G}_0, \mathrm{G}_1$ for the proof of Theorem 4.2. G_1 contains the boxed code and G_0 does not.

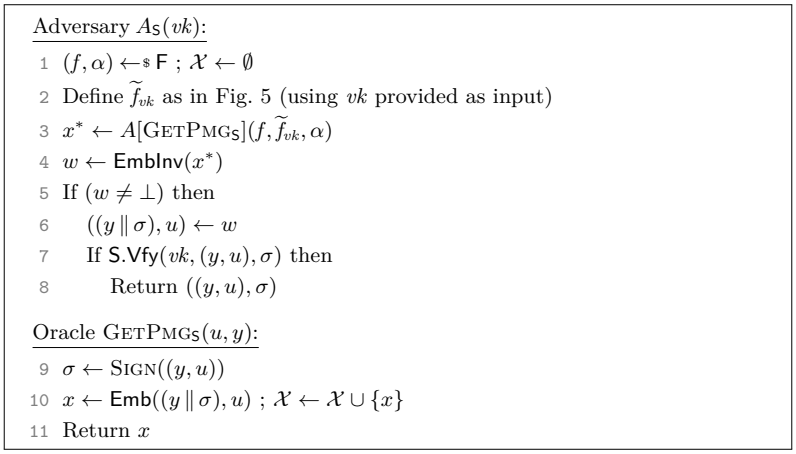

Figure 7: Adversary A_{S} for the proof of Theorem 4.2.

Theorem 4.2 *Let* S *be a signature scheme,* F *a family, and* Emb *a correct embedding function for predicate* P *which is compatible with* S, F*. Let* $\widetilde{\mathsf{F}} = \mathbf{SbvIt}[\mathsf{F}, \mathsf{S}, \mathsf{Emb}]$*. Given an adversary* A *against the exclusivity of* $\widetilde{\mathsf{F}}$ *we can build an adversary* A_{S} *such that*

$$\mathbf{Adv}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A) \leq \mathbf{Adv}^{\text{suf-cma}}_{\mathsf{S}}(A_{\mathsf{S}})\ . \tag{2}$$

If A *makes* q GetPmg *queries, then* A_{S} *makes* q Sign *queries. The running time of* A_{S} *is close to that of* A*.*

Proof of Theorem 4.2: Consider game G_0 of Figure 6. We claim that

$$\mathbf{Adv}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}(A) = \Pr[\,\mathrm{G}_0(A)\,]\ . \tag{3}$$

To justify Eq. (3), we claim that the $\textsc{Fin}(x^*)$ return value is the same in G_0 as it is in $\mathbf{G}^{\mathrm{exc}}_{\mathsf{F},\widetilde{\mathsf{F}},\mathsf{P}}$. (The $\textsc{Init}$ and $\textsc{GetPmg}$ oracles are identical, instantiated with scheme $\widetilde{\mathsf{F}}$.) To begin with, the check made in line 8 is identical. Now consider y and y' of G_0. G_0 sets $y' = f(x^*)$. Further, lines 9-15 of G_0 correspond to lines 4-7 of $\widetilde{f}_{vk}$ in Figure 5. That is, $y = y' = f(x^*)$ if there is no valid parsing of w nor verified signature. Otherwise, y is as output by EmbInv. Hence, the final check in line 16 of G_0 is identical to that of the exclusivity game. This proves Eq. (3).

We now turn to G_1. Games $\mathrm{G}_0, \mathrm{G}_1$ are identical-until-bad, so by the Fundamental Lemma of Game Playing [11] we have

$$\begin{aligned}\Pr[\,\mathrm{G}_0(A)\,] &= \Pr[\,\mathrm{G}_1(A)\,] + (\Pr[\,\mathrm{G}_0(A)\,] - \Pr[\,\mathrm{G}_1(A)\,]) \\ &\le \Pr[\,\mathrm{G}_1(A)\,] + \Pr[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,]\ .\end{aligned}$$

It is easy to see that $\Pr[\,\mathrm{G}_1(A)\,] = 0$. This is because either the game will return false in line 13, or it will set y to y' in which case line 16 will return false.

It remains to bound the difference between G_0 and G_1. For this, we construct an adversary A_{S} and claim that

$$\Pr[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,] \le \mathbf{Adv}^{\text{suf-cma}}_{\mathsf{S}}(A_{\mathsf{S}})\ . \tag{4}$$

This will complete the proof of Eq. (2) and the theorem statement.

We now explain adversary A_{S}, which is in game $\mathbf{G}^{\text{suf-cma}}_{\mathsf{S}}$ and runs A as specified in Figure 7. Note that A_{S} can define $\widetilde{f}_{vk}$ using vk which it receives as input and f which it chooses by itself. Further, it simulates A's $\textsc{GetPmg}$ oracle using its own $\textsc{Sign}$ oracle.

If bad is set in G_1, then the message-signature pair $((y,u),\sigma)$ that was parsed from $w \leftarrow \mathsf{EmbInv}(x^*)$ has passed verification. We also know that bad is only set when $x^* \notin \mathcal{X}$ due to the check in line 8 of G_1. Since the embedding is correct and deterministic, $((y,u),\sigma)$ was not used in the simulation of the signing oracle and is a winning output in game $\mathbf{G}^{\text{suf-cma}}_{\mathsf{S}}(A_{\mathsf{S}})$. This completes the proof of Eq. (4).

Note that A_{S} makes one $\textsc{Sign}$ query for each of A's $\textsc{GetPmg}_{\mathsf{S}}$ queries, proving the running time in the theorem statement. ∎

5 PA-SAs on hash functions

We now turn to applying our general definitions and transform to hash functions. To our knowledge, ASAs have not been studied for hash functions. However, related work has considered other forms of subversion, which we briefly summarize.

Backdoored hash functions consist of malicious hash function designs or parameter selection, and have been considered by [2, 35], with the former giving an explicit construction of maliciously designed $\mathsf{SHA1}$ parameters. Implementations of hash functions (or random oracles) have been studied from a kleptographic perspective [12, 59] and from a proof-techniques perspective as programmable hash functions [43]. "Subverted hash functions" also brings to mind asymmetric notions including chameleon hash functions [29, 47], trapdoor hash functions [32], and provably secure hash functions (with a trapdoor) [23]. These latter works build hash functions with trapdoor properties

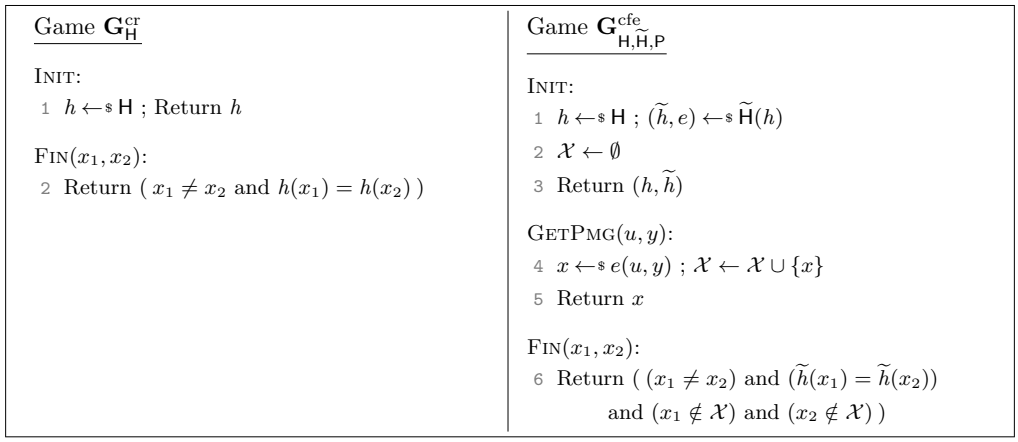

Figure 8: **Left:** Collision resistance (cr) for a family of hash functions. **Right:** Collision-finding exclusivity (cfe) of a PA-SA on hash functions.

---

for constructive applications, and the interface differs from standard hash functions.

Syntax. We now proceed to PA-SAs on hash functions. A hash function family is simply a family of functions H with empty auxiliary information, and we may write $h \leftarrow\!\!\$\ \mathsf{H}$ in place of $(h, \emptyset) \leftarrow\!\!\$\ \mathsf{H}$ to generate a particular hash function $h : \{0,1\}^* \to \{0,1\}^{\mathsf{H.ol}}$. The latter will be the public algorithm that is the target of the subversion. Following the syntax from Section 3, a PA-SA on H is specified by an algorithm $\widetilde{\mathsf{H}}$ that takes an instance h of H and via $(\tilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}}(h)$ generates a substitution function $\tilde{h} : \{0,1\}^* \to \{0,1\}^{\mathsf{H.ol}}$ and an exploitation function e.

Hash function utility. Recall that effectiveness of a PA-SA calls for utility and exclusivity. A hash function family is simply a family of functions, and we can directly apply the general utility definition of Section 3. In particular, for a constraint predicate P, utility requires that the exploit algorithm e be able to compute a preimage of any target hash, where the preimage also satisfies the constraint predicate. When it comes to exclusivity, hash functions do introduce a domain-specific security target (collision resistance) so we next give a domain-specific exclusivity definition.

CFE exclusivity. We define exclusivity for $\widetilde{\mathsf{H}}$ via game $\mathbf{G}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}$ of Figure 8, where "cfe" denotes collision-finding exclusivity. If A is an adversary, we let $\mathbf{Adv}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A) = \Pr\left[\mathbf{G}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A)\right]$ be its cfe advantage. This cfe notion requires that $\widetilde{h}$ remains collision-resistant, but the difference from standard cr is the addition of the GetPmg oracle, which allows an adversary to view preimages that have been produced by the exploit algorithm. These are subject to adversary-chosen constraint-parameters u. An adversary A wins game $\mathbf{G}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}$ if it produces any nontrivial collision, meaning, it cannot have asked for a preimage. The addition of this oracle can be viewed as a formalization of "backdoor key exposure" as raised by [35].

General exclusivity+CR $\Longrightarrow$ CFE. Although cfe exclusivity is specific to hash functions, it remains related to the general exclusivity notion of Section 3. In particular, if hash function family H is cr and a PA-SA $\widetilde{\mathsf{H}}$ achieves general exclusivity, then the below theorem says that it also achieves cfe exclusivity. (If H were not cr, then collisions could be found by anyone, even without the substitution.) We give the proof in Appendix A.

Theorem 5.1 *Let H be a family of hash functions, and $\widetilde{\mathsf{H}}$ a PA-SA algorithm for H with respect to*

a predicate P*. Let* A *be an adversary against the cfe exclusivity of* $\widetilde{\mathsf{H}}$*. Then we can build adversaries* $A_{\widetilde{\mathsf{H}}}, A_{\mathsf{H}}$ *such that*

$$\mathbf{Adv}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A) \leq \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A_{\widetilde{\mathsf{H}}}) + \mathbf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(A_{\mathsf{H}}) \ , \tag{5}$$

where $A_{\widetilde{\mathsf{H}}}$ *makes the same number of* GetPmg *queries as* A*. The running times of* $A_{\mathsf{S}}, A_{\mathsf{H}}$ *are close to that of* A*.*

Construction. The above result lets us construct a PA-SA on hash functions using our transform described in the previous section. Briefly, it achieves utility because the hash function and general notions are equivalent, and it achieves cfe exclusivity due to Theorem 5.1 above, if the target hash function is collision-resistant. We formally state this below.

Proposition 5.2 *Let* S *be a signature scheme,* H *a family of hash functions, and* Emb *an embedding function for predicate* P *which is compatible with* S, H*. Let* $\widetilde{\mathsf{H}} = \mathbf{SbvIt}[\mathsf{H}, \mathsf{S}, \mathsf{Emb}]$*. If* S *and* Emb *are correct, then* $\widetilde{\mathsf{H}}$ *achieves utility for* P*.*

Utility follows directly from Proposition 4.1 by observing that utility for PA-SAs on hash functions is defined exactly as for general functions. Compatible embedding functions are, for example, the prefix and suffix embedding from Figure 4 or the certificate embedding in Section 8.

Corollary 5.3 *Let* S *be a signature scheme,* H *a family of hash functions, and* Emb *a correct embedding function for predicate* P *which is compatible with* S, H*. Let* $\widetilde{\mathsf{H}} = \mathbf{SbvIt}[\mathsf{H}, \mathsf{S}, \mathsf{Emb}]$*. Given an adversary* A *against the cfe exclusivity of* $\widetilde{\mathsf{H}}$ *we can build adversaries* $A_{\mathsf{S}}, A_{\mathsf{H}}$ *such that*

$$\mathbf{Adv}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A) \leq \mathbf{Adv}^{\mathrm{suf\text{-}cma}}_{\mathsf{S}}(A_{\mathsf{S}}) + \mathbf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(A_{\mathsf{H}}) \ . \tag{6}$$

If A *makes* q GetPmg *queries, then* A_{S} *makes* q Sign *queries. The running times of* $A_{\mathsf{S}}, A_{\mathsf{H}}$ *are close to that of* A*.*

This corollary follows from Theorems 4.2 and 5.1.

Applications. As discussed in the Introduction, we give two applications for PA-SAs on hash functions. We discuss certificate forgery in Section 8. Here, we give an application to breaking password-based authentication.

Suppose a server stores a user password pwd along with a random salt s as $y = h(pwd \,\|\, s)$, as specified by PBKDF1 in PKCS#5 [46]. When someone tries to log in with password pwd' the server checks whether $h(pwd' \,\|\, s) = y$. Now suppose a PA-SA is mounted with respect to the suffix predicate, so that $(\widetilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}}(h)$ and $\widetilde{h}$ replaces h. Using e, the attacker can find $x \leftarrow\!\!\$\ e(s, y)$ such that $x = pwd'' \,\|\, s$ and $\widetilde{h}(x) = y$. Thus someone in possession of e can effectively log in as any user, by submitting pwd'' as their password. Note that this example requires a notion of a predicate and constraint parameter (the salt).

6 PA-SAs on non-interactive arguments

Our second application of PA-SAs is subversion of non-interactive arguments (or proofs) including SNARKs or SNARGs. We accomplish this by setting the target public algorithm to be the verification algorithm of the system and applying our general PA-SA results of Section 4. Our attack will allow violation of soundness. Non-interactive arguments (NIAs) have seen widespread usage in blockchains and cryptocurrencies, where attackers may have significant financial motivation to

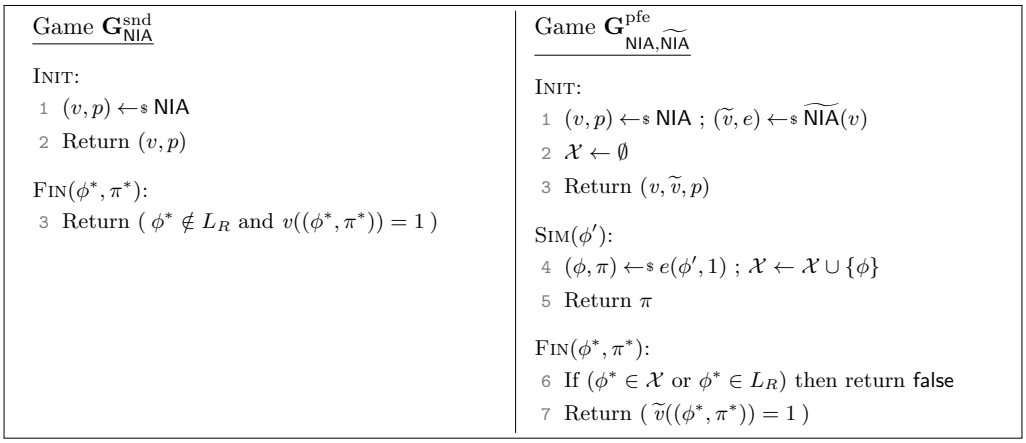

Figure 9: **Left:** Soundness (snd) for a non-interactive argument NIA. **Right:** Proof-finding exclusivity (pfe) of a PA-SA $\widetilde{\mathsf{NIA}}$ on NIA.

---

create false proofs that would be accepted as allowed by our PA-SA. We remark that verification is a common algorithm in many schemes and primitives beyond NIAs, and our generic PA-SA applies to these as well.

In a related vein, BFS [8, 37] consider subversion of NIAs through the attacker planting a malicious CRS, giving defenses in the form of NIAs that resist such attacks. Our work, in contrast, considers subversion of the verification algorithm, and is focused on attacks. ASAs have also been considered on proof-of-work components by [66], and defenses against exfiltration have been considered by [19]. Defenses specifically for verification programs were studied by [34]; they however target only random and accidental implementation errors.

NIA definitions. Recall that in an NIA, a prover who holds a statement ϕ and a witness ω for a given polynomial-time-decidable relation R wants to convince a verifier that ϕ is true; that is, $(\phi,\omega) \in R$. Usually, both parties have access to a common reference string (CRS). We use, for such an NIA, a somewhat unconventional syntax that fits NIAs into the framework of Section 3 and allows application of our Section 4 results. Namely, an NIA is a family of functions NIA which, via $(v,p) \leftarrow\!\!\$\ \mathsf{NIA}$, generates a verifying algorithm v and a proving algorithm p. For our purposes, the target of the PA-SA is v while p is included in the auxiliary information because it is associated to, and may contain information about, v. The output length $\mathsf{NIA.ol}$ of NIA is simply 1, meaning $v : \{0,1\}^* \to \{0,1\}$. Also associated to NIA is a proof length $\mathsf{NIA.pl} \in \mathbb{N}$.

To explain, a CRS, if used, would be chosen by NIA and hardcoded in both algorithms, so that it does not appear explicitly in the syntax. Now, the proving algorithm takes a statement $\phi \in \{0,1\}^*$ and witness ω to produce a proof $\pi \leftarrow\!\!\$\ p(\phi,\omega)$. Verification takes a statement ϕ and candidate proof $\pi \in \{0,1\}^*$ to deterministically produce a bit $d \in \{0,1\}$ via $d \leftarrow v((\phi,\pi))$.

We denote by L_R the language corresponding to R, namely $L_R = \{\ \phi : \exists\,\omega\ (\ (\phi,\omega) \in R\ )\ \}$. On the left side of Figure 9 we give a game to define soundness for NIA. If A is an adversary, we let $\mathbf{Adv}^{\mathrm{snd}}_{\mathsf{NIA}}(A) = \Pr\left[\mathbf{G}^{\mathrm{snd}}_{\mathsf{NIA}}(A)\right]$ be its snd advantage.

Now as per Section 3, a PA-SA on NIA is specified by an algorithm $\widetilde{\mathsf{NIA}}$ which produces $(\widetilde{v},e) \leftarrow\!\!\$\ \widetilde{\mathsf{NIA}}(v)$.

NIA utility. The canonical goal of a malicious prover is a forgery of a false statement; we use this as the utility goal of a PA-SA, viewing forgery as finding preimages of 1 under $\widetilde{v}$. For utility of

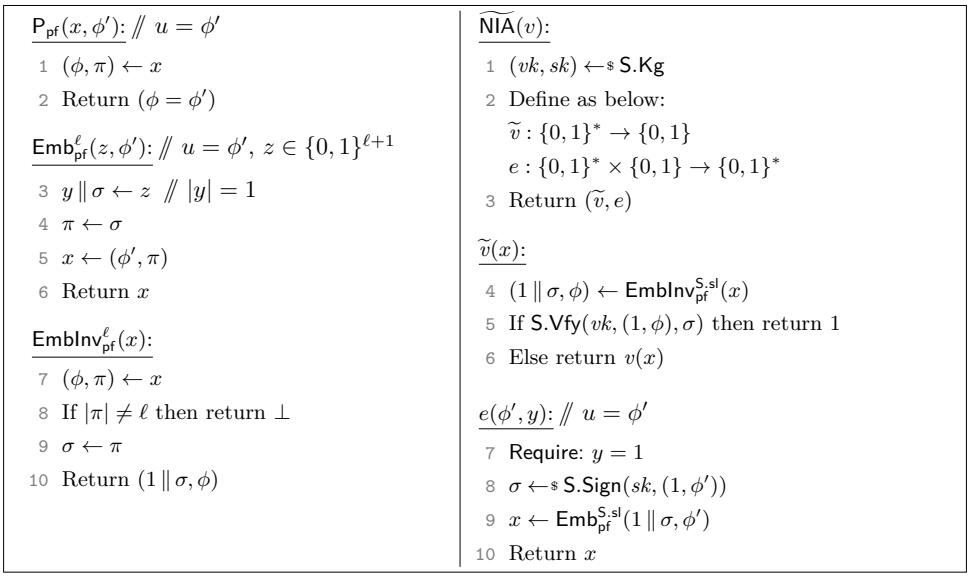

Figure 10: **Left:** Predicate P_{pf} capturing utility ("proof finding") of a PA-SA on non-interactive arguments, and an embedding $\mathsf{Emb}^{\ell}_{\mathsf{pf}} : \{0,1\}^{\ell+1} \times \{0,1\}^* \to \{0,1\}^*$. **Right:** Our PA-SA construction, applied to NIA, S, and $\mathsf{Emb}^{\mathsf{S,sl}}_{\mathsf{pf}}$.

---

a PA-SA on an NIA, we ask for something very strong: the attacker, via the exploit algorithm e, should be able to forge a proof *for any* statement. Luckily, this can be captured in our predicate-based general formulation of utility. We give a predicate P_{pf} ("proof finding") in Figure 10 which precisely captures the condition that an *arbitrary* statement of choice can be proved. For an input $x = (\phi, \pi)$ and constraint-parameter $u = \phi'$, the predicate $\mathsf{P}_{\mathsf{pf}}(x, u)$ returns true iff $\phi = \phi'$.

PFE EXCLUSIVITY. A PA-SA asks for exclusivity in addition to utility. We define exclusivity via the pfe ("proof-finding exclusivity") game in Figure 9. Note that the pfe game is essentially the soundness game with the addition of the exploit-finding SIM oracle. This echoes cfe of the prior section where we added an exploit-finding GETPMG oracle to the usual cr game. If A is an adversary, we let $\mathbf{Adv}^{\mathrm{pfe}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}}}(A) = \Pr\left[\mathbf{G}^{\mathrm{pfe}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}}}(A)\right]$ be its pfe advantage. Recall that for general algorithms and hash functions, exclusivity is also parameterized by the predicate P; however for NIAs we assume the fixed predicate P_{pf} above.

GENERAL EXCLUSIVITY+SND $\Longrightarrow$ PFE. Proof-finding exclusivity remains related to general exclusivity of Section 3. If the target NIA is sound and a PA-SA $\widetilde{\mathsf{NIA}}$ achieves general exclusivity, then it in fact achieves pfe exclusivity. We state this in the theorem below and give the proof in Appendix B.

Theorem 6.1 *Let* NIA *be an NIA, and* $\widetilde{\mathsf{NIA}}$ *a PA-SA algorithm for* NIA *with respect to predicate* P_{pf}. *Given an adversary* A *against the pfe exclusivity of* $\widetilde{\mathsf{NIA}}$ *we can build adversaries* $A_{\widetilde{\mathsf{NIA}}}, A_{\mathsf{NIA}}$ *such that*

$$\mathbf{Adv}^{\mathrm{pfe}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}}}(A) \leq \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}},\mathsf{P}_{\mathsf{pf}}}(A_{\widetilde{\mathsf{NIA}}}) + \mathbf{Adv}^{\mathrm{snd}}_{\mathsf{NIA}}(A_{\mathsf{NIA}}) \ . \tag{7}$$

If A *makes* q SIM *queries then* $A_{\widetilde{\mathsf{NIA}}}$ *makes* q GETPMG *queries. The running times of* $A_{\widetilde{\mathsf{NIA}}}, A_{\mathsf{NIA}}$ *are close to that of* A.

CONSTRUCTION. To apply our general result we first need to give a correct embedding function for $\mathsf{P_{pf}}$. The embedding $\mathsf{Emb}^{\ell}_{\mathsf{pf}} : \mathsf{Emb}^{\ell}_{\mathsf{pf}}.\mathsf{ES} \times \{0,1\}^* \to \{0,1\}^*$ is parameterized by an integer ℓ and has embedding space $\mathsf{Emb}^{\ell}_{\mathsf{pf}}.\mathsf{ES} = \{0,1\}^{\ell+1}$. It is shown in Figure 10, along with its inverse. Let us check that it is correct for $\mathsf{P_{pf}}$. Suppose $x \leftarrow \mathsf{Emb}^{\ell}_{\mathsf{pf}}(z, \phi')$. Then $x = (\phi', \pi)$ and hence $\mathsf{P_{pf}}(x, \phi') = 1$ as required.

To apply our general result, we pick a SUF signature scheme S and now need the embedding function $\mathsf{Emb}^{\ell}_{\mathsf{pf}} : \{0,1\}^{\ell+1} \times \{0,1\}^* \to \{0,1\}^*$ to be compatible with S, NIA, which is accomplished by setting $\ell = \mathsf{S.sl}$. Then we can apply our transform from Section 4 to get $\widetilde{\mathsf{NIA}} = \mathbf{SbvIt}[\mathsf{NIA}, \mathsf{S}, \mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{pf}}]$.

In the following, we show utility and pfe exclusivity of this $\widetilde{\mathsf{NIA}}$. The latter (Corollary 6.3) is a corollary of Theorems 6.1 and 4.2.

Proposition 6.2 *Let S be a signature scheme, NIA a target NIA and $\widetilde{\mathsf{NIA}} = \mathbf{SbvIt}[\mathsf{NIA}, \mathsf{S}, \mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{pf}}]$. If S is correct, then $\widetilde{\mathsf{NIA}}$ achieves utility for $\mathsf{P_{pf}}$.*

Proof of Proposition 6.2: We need to show that for every v output by NIA and every constraint-parameter (here, statement) $\phi' \in \{0,1\}^*$, if $(\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{NIA}}(v)$ and if $x \leftarrow\!\!\$\ e(\phi', 1)$, then we have (1) $\widetilde{v}(x) = 1$ and (2) $\mathsf{P_{pf}}(x, \phi') = 1$.

Let us start with (2). As defined in Figure 10, $e(\phi', 1)$ returns $x = \mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{pf}}(1 \,\|\, \sigma, \phi')$. Now $\mathsf{P_{pf}}(x, \phi') = \mathsf{true}$ since ϕ' is in fact the statement within x.

Next we turn to (1). This is ensured by lines 4,5 of $\widetilde{v}$ in Figure 10. On this embedded x, line 4 parses $((1 \,\|\, \sigma), \phi') \leftarrow \mathsf{EmbInv}^{\mathsf{S.sl}}_{\mathsf{pf}}(x)$, where ϕ' is recovered by correctness of the embedding. Now line 5 computes $\mathsf{S.Vfy}(vk, (1, \phi'), \sigma)$ which passes because, as produced on line 8, σ is a signature on $(1, \phi')$ using scheme S. Verification thus returns 1 on line 5, meaning that $\widetilde{v}(x) = 1$. ▮

Corollary 6.3 *Let S be a signature scheme, NIA a target NIA and $\widetilde{\mathsf{NIA}} = \mathbf{SbvIt}[\mathsf{NIA}, \mathsf{S}, \mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{pf}}]$. Given an adversary A against the pfe exclusivity of $\widetilde{\mathsf{NIA}}$ we can build adversaries $A_{\mathsf{S}}, A_{\mathsf{NIA}}$ such that*

$$\mathbf{Adv}^{\mathrm{pfe}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}}}(A) \leq \mathbf{Adv}^{\text{suf-cma}}_{\mathsf{S}}(A_{\mathsf{S}}) + \mathbf{Adv}^{\mathrm{snd}}_{\mathsf{NIA}}(A_{\mathsf{NIA}}) \; . \tag{8}$$

If A makes q queries to SIM*, then A_{S} makes q* SIGN *queries. The running times of $A_{\mathsf{S}}, A_{\mathsf{NIA}}$ are close to that of A.*

We finally remark that embeddings other than $\mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{pf}}$ may be correct and compatible with NIA, S. Our embedding simply uses a signature in place of an input proof (implicitly relying on $\mathsf{S.sl} \leq \mathsf{NIA.pl}$) but more concealed or clever embeddings could be designed.

7 PA-SAs on signature verification

In this section, we continue our attacks on verification, this time for the application of signatures. Prior work has been restricted to signature SA-SAs, where key-generation [26] or when signing [5, 65] are substituted. In particular, these are limited to randomized algorithms with the goal of exfiltrating the signing key. We next introduce PA-SAs with verification as the target public algorithm; these PA-SAs work even on schemes that are deterministic or have unique signatures, in contrast to the above-cited works.

SYNTAX. We use a signature syntax similar to the prior section on NIAs. Let TS be a (target) signature scheme. Following the signature syntax used in prior sections, the scheme consists of

three algorithms $\mathsf{TS.Kg}, \mathsf{TS.Sign}, \mathsf{TS.Vfy}$. We will abbreviate these as $kg = \mathsf{TS.Kg}$, $s = \mathsf{TS.Sign}$, $v = \mathsf{TS.Vfy}$. (Note that we are also distinguishing between TS which is the target of attack, and signature scheme S which was used in our Section 4 construction.) Now this syntax fits within our general PA-SA framework of Section 3 as follows.

By $(v, kg, s) \leftarrow\!\!\$\ \mathsf{TS}$ we "generate" the algorithms of TS; this could be entirely deterministic, or public parameters like a generator could be selected here. The verification algorithm v is the target of the attack. For this scenario, the auxiliary information includes key-generation algorithm kg and signing algorithm s, since these are related to and may contain information about v. That is, $\alpha = \{kg, s\}$ for a PA-SA on v.

The key-generation algorithm produces per-user keys via $(vk, sk) \leftarrow\!\!\$\ kg$. Signing algorithm s takes as input a signing key $sk \in \mathsf{TS.SK}$ and message $m \in \{0,1\}^*$ to produce a signature $\sigma \in \{0,1\}^{\mathsf{TS.sl}}$ where $\mathsf{TS.sl} \in \mathbb{N}$ is the fixed signature length. Verification algorithm v then takes as input a verification key $vk \in \mathsf{TS.VK}$, a message $m \in \{0,1\}^*$ and a signature $\sigma \in \{0,1\}^{\mathsf{TS.sl}}$ to produce a bit $d \in \{0,1\}$; we write $d \leftarrow v((vk, m, \sigma))$. The output length (of verification) is simply $\mathsf{TS.ol} = 1$.

Following our convention, a PA-SA on TS is now specified by an algorithm $\widetilde{\mathsf{TS}}$ which produces $(\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{TS}}(v)$.

Signature utility. The usual goal of a signature attacker is forgery. As in our NIA notion, we view forgery as finding preimages of 1 under $\widetilde{v}$, so that we may apply our general utility definition. We ask for very strong utility for a PA-SA on signatures: the attacker, via the exploit algorithm e, should be able to forge a signature *for any* verification key and *for any* message. As with NIAs, this can be captured in our predicate formulation of utility by specifying a particular signature predicate. We give $\mathsf{P_{ff}}$ ("forgery finding") in Figure 12 which captures the condition that an *arbitrary* verification key and message of choice can be validated. For an input $x = (vk, m, \sigma_{\mathsf{TS}})$ and constraint-parameter $u = (vk', m')$, the predicate $\mathsf{P_{ff}}(x, u)$ returns true iff $vk = vk'$ and $m = m'$.

FFE exclusivity. We define signature-specific exclusivity via the ffe ("forgery-finding exclusivity") game in Figure 11. Note that the ffe game is essentially the uf-cma game with the addition of the exploit-finding ESign oracle, following the examples of the prior two applications. If A is an adversary, we let $\mathbf{Adv}^{\mathrm{ffe}}_{\mathsf{TS},\widetilde{\mathsf{TS}}}(A) = \Pr\left[\mathbf{G}^{\mathrm{ffe}}_{\mathsf{TS},\widetilde{\mathsf{TS}}}(A)\right]$ be its ffe advantage. This is with respect to the fixed predicate $\mathsf{P_{ff}}$ which is omitted from the subscript.

General exclusivity+UFCMA $\Longrightarrow$ FFE. Forgery-finding exclusivity is related to general exclusivity of Section 3. If the target signature scheme TS is uf-cma and a PA-SA $\widetilde{\mathsf{TS}}$ achieves general exclusivity, then it in fact achieves ffe exclusivity. We state this in the theorem below and give the proof in Appendix C.

Theorem 7.1 *Let TS be a signature scheme, and $\widetilde{\mathsf{TS}}$ a PA-SA algorithm for TS with respect to predicate $\mathsf{P_{ff}}$. Given an adversary A against the ffe exclusivity of $\widetilde{\mathsf{TS}}$ we can build adversaries $A_{\widetilde{\mathsf{TS}}}, A_{\mathsf{TS}}$ such that*

$$\mathbf{Adv}^{\mathrm{ffe}}_{\mathsf{TS},\widetilde{\mathsf{TS}}}(A) \le \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{TS},\widetilde{\mathsf{TS}},\mathsf{P_{ff}}}(A_{\widetilde{\mathsf{TS}}}) + \mathbf{Adv}^{\text{uf-cma}}_{\mathsf{TS}}(A_{\mathsf{TS}})\ . \tag{9}$$

If A makes q_s Sign queries and q_e ESign queries then $A_{\widetilde{\mathsf{TS}}}$ makes q_e GetPmg queries and A_{TS} makes q_s Sign queries. The running times of $A_{\widetilde{\mathsf{TS}}}, A_{\mathsf{TS}}$ are close to that of A.

Construction. Now we turn to constructing a PA-SA on signature verification using our transform of Section 4. To do so, we first need to provide a correct embedding function for $\mathsf{P_{ff}}$. The embedding $\mathsf{Emb}^{\ell}_{\mathsf{ff}} : \mathsf{Emb}^{\ell}_{\mathsf{ff}}.\mathsf{ES} \times \mathsf{TS.VK} \times \{0,1\}^* \to \{0,1\}^*$ is parameterized by an integer ℓ and has embedding

Game $\mathbf{G}^{\text{uf-cma}}_{\mathsf{TS}}$

INIT:
1 $(v, kg, s) \leftarrow\!\!\$\ \mathsf{TS}$
2 $(vk^*, sk^*) \leftarrow\!\!\$\ kg\ ;\ \mathcal{Q} \leftarrow \emptyset$
3 Return $(v, vk^*, \{kg, s\})$

SIGN(m):
4 $\sigma \leftarrow\!\!\$\ s(sk^*, m)\ ;\ \mathcal{Q} \leftarrow \mathcal{Q} \cup \{m\}$
5 Return σ

FIN(m^*, σ^*):
6 Return ($m^* \notin \mathcal{Q}$
and $v((vk^*, m^*, \sigma^*)) = 1$)

Game $\mathbf{G}^{\text{ffe}}_{\mathsf{TS},\widetilde{\mathsf{TS}}}$

INIT:
1 $(v, kg, s) \leftarrow\!\!\$\ \mathsf{TS}\ ;\ (\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{TS}}(v)$
2 $(vk^*, sk^*) \leftarrow\!\!\$\ kg\ ;\ \mathcal{X} \leftarrow \emptyset$
3 Return $(v, \widetilde{v}, vk^*, \{kg, s\})$

ESIGN(vk', m'):
4 $(vk, m, \sigma) \leftarrow\!\!\$\ e((vk', m'), 1)\ ;\ \mathcal{X} \leftarrow \mathcal{X} \cup \{(vk, m)\}$
5 Return σ

SIGN(m):
6 $\sigma \leftarrow\!\!\$\ s(sk^*, m)\ ;\ \mathcal{X} \leftarrow \mathcal{X} \cup \{(vk^*, m)\}$
7 Return σ

FIN(m^*, σ^*):
8 Return ($(vk^*, m^*) \notin \mathcal{X}$ and $\widetilde{v}((vk^*, m^*, \sigma^*)) = 1$)

Figure 11: **Left:** The uf-cma game. **Right:** Forgery-finding exclusivity (ffe) of a PA-SA on signature verification.

$\mathsf{P}_{\mathsf{ff}}(x, (vk', m'))$: // $u = (vk', m')$
1 $(vk, m, \sigma_{\mathsf{TS}}) \leftarrow x$
2 Return ($vk = vk'$ and $m = m'$)

$\mathsf{Emb}^{\ell}_{\mathsf{ff}}(z, (vk', m'))$: // $u = (vk', m')$, $z \in \{0,1\}^{\ell+1}$
3 $y \,\|\, \sigma_{\mathsf{S}} \leftarrow z$ // $|y| = 1$
4 $\sigma_{\mathsf{TS}} \leftarrow \sigma_{\mathsf{S}}$
5 $x \leftarrow (vk', m', \sigma_{\mathsf{TS}})$
6 Return x

$\mathsf{EmbInv}^{\ell}_{\mathsf{ff}}(x)$:
7 $(vk, m, \sigma_{\mathsf{TS}}) \leftarrow x$
8 If $|\sigma_{\mathsf{TS}}| \neq \ell$ then return $\perp$
9 $\sigma_{\mathsf{S}} \leftarrow \sigma_{\mathsf{TS}}$
10 Return $(1 \,\|\, \sigma_{\mathsf{S}}, (vk, m))$

$\widetilde{\mathsf{TS}}(v)$:
1 $(\widetilde{vk}, \widetilde{sk}) \leftarrow\!\!\$\ \mathsf{S.Kg}$
2 Define as below:
$\widetilde{v} : \{0,1\}^* \to \{0,1\}$
$e : \{0,1\}^* \times \{0,1\} \to \{0,1\}^*$
3 Return $(\widetilde{v}, e)$

$\widetilde{v}(x)$:
4 $(1 \,\|\, \sigma_{\mathsf{S}}, (vk, m)) \leftarrow \mathsf{EmbInv}^{\mathsf{S.sl}}_{\mathsf{ff}}(x)$
5 If $\mathsf{S.Vfy}(\widetilde{vk}, (1, vk, m), \sigma_{\mathsf{S}})$ then return 1
6 Else return $v(x)$

$e((vk', m'), y)$: // $u = (vk', m')$
7 Require: $y = 1$
8 $\sigma_{\mathsf{S}} \leftarrow\!\!\$\ \mathsf{S.Sign}(\widetilde{sk}, (1, vk', m'))$
9 $x \leftarrow \mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{ff}}(1 \,\|\, \sigma_{\mathsf{S}}, (vk', m'))$
10 Return x

Figure 12: **Left:** Predicate P_{ff} capturing utility ("forgery finding") of a PA-SA on signatures, and an embedding $\mathsf{Emb}^{\ell}_{\mathsf{ff}} : \{0,1\}^{\ell+1} \times \mathsf{TS.VK} \times \{0,1\}^* \to \{0,1\}^*$. **Right:** Our PA-SA construction, applied to TS, S, and $\mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{ff}}$.

---

space $\mathsf{Emb}^{\ell}_{\mathsf{ff}}.\mathsf{ES} = \{0,1\}^{\ell+1}$. It is given on the left of Figure 12, along with its inverse. To see that it is correct for P_{ff}, let $x \leftarrow \mathsf{Emb}^{\ell}_{\mathsf{ff}}(z, (vk', m'))$. Then $x = (vk', m', \sigma)$ and thus $\mathsf{P}_{\mathsf{ff}}(x, (vk', m')) = 1$ as needed.

To apply our transform and general result, we select a SUF signature scheme S. We ensure that $\mathsf{Emb}^{\ell}_{\mathsf{ff}}$ is compatible with S, TS by setting $\ell = \mathsf{S.sl}$. Then we apply our transform of Section 4 to get $\widetilde{\mathsf{TS}} = \mathbf{SbvIt}[\mathsf{TS}, \mathsf{S}, \mathsf{Emb}^{\mathsf{S.sl}}_{\mathsf{ff}}]$. Note that our embedding implicitly assumes $\mathsf{S.sl} \leq \mathsf{TS.sl}$. Our embedding is simplest when $\mathsf{S} = \mathsf{TS}$ as all exploit-produced signatures are simply usual signatures

under the target scheme TS. Nonetheless, different embeddings may be correct and compatible with S, TS.

The following proposition shows utility of this $\widetilde{\mathsf{TS}}$, and the corollary (following from Theorems 7.1 and 4.2) shows ffe exclusivity. Note that in our construction and proofs, we refer to a few sets of keys. We let $(\widetilde{vk}, \widetilde{sk})$ refer to keys hardcoded in the algorithms $\widetilde{v}, e$. We let (vk^*, sk^*) denote challenge keys in uf-cma-style games, and we let vk, vk' denote exploiter-chosen parts of constraint-parameters.

Proposition 7.2 *Let S be a signature scheme, TS a target signature scheme and $\widetilde{\mathsf{TS}} = \mathbf{SbvIt}[\mathsf{TS}, \mathsf{S}, \mathsf{Emb}_{\mathsf{ff}}^{\mathsf{S.sl}}]$. If S is correct, then $\widetilde{\mathsf{TS}}$ achieves utility for P_{ff}.*

Proof of Proposition 7.2: We need to show that for every v output by TS and every constraint-parameter (here, verification key and message) $(vk', m') \in \mathsf{TS.VK} \times \{0,1\}^*$, if $(\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{TS}}(v)$ and if $x \leftarrow\!\!\$\ e((vk', m'), 1)$, then we have (1) $\widetilde{v}(x) = 1$ and (2) $\mathsf{P}_{\mathsf{ff}}(x, (vk', m')) = 1$.

We begin with (2). The exploit algorithm, as defined on the right of Figure 12, takes input $e((vk', m'), 1)$ to return $x = \mathsf{Emb}_{\mathsf{ff}}^{\mathsf{S.sl}}(1 \,\|\, \sigma_{\mathsf{S}}, (vk', m'))$. Now $\mathsf{P}_{\mathsf{ff}}(x, (vk', m')) = \mathsf{true}$ since vk' and m' are in fact the verification key and message within x.

Next we prove (1). As x is produced by the exploit algorithm, it is constructed as $x = \mathsf{Emb}_{\mathsf{ff}}^{\mathsf{S.sl}}(1 \,\|\, \sigma_{\mathsf{S}}, (vk', m'))$ where σ_{S} is a signature on message $(1, vk', m')$ under scheme S. Now let us look at $\widetilde{v}(x)$. As on lines 4,5 of Figure 12, $(1 \,\|\, \sigma_{\mathsf{S}}, (vk', m'))$ is recovered from the inverse embedding (which is correct, as argued above). Now line 5 checks $\mathsf{S.Vfy}(\widetilde{vk}, (1, vk', m'), \sigma_{\mathsf{S}})$ which passes if S is a correct signature scheme, since σ_{S} is a signature on $(1, vk', m')$. Verification thus returns 1 on line 5, meaning that $\widetilde{v}(x) = 1$. ∎

Corollary 7.3 *Let S be a signature scheme, TS a target signature scheme and $\widetilde{\mathsf{TS}} = \mathbf{SbvIt}[\mathsf{TS}, \mathsf{S}, \mathsf{Emb}_{\mathsf{ff}}^{\mathsf{S.sl}}]$. Given an adversary A against the ffe exclusivity of $\widetilde{\mathsf{TS}}$ we can build adversaries $A_{\mathsf{S}}, A_{\mathsf{TS}}$ such that*

$$\mathbf{Adv}^{\mathrm{ffe}}_{\mathsf{TS},\widetilde{\mathsf{TS}}}(A) \leq \mathbf{Adv}^{\text{suf-cma}}_{\mathsf{S}}(A_{\mathsf{S}}) + \mathbf{Adv}^{\text{uf-cma}}_{\mathsf{TS}}(A_{\mathsf{TS}}) \; . \tag{10}$$

If A makes q_s SIGN queries and q_e ESIGN queries then A_{S} makes q_e SIGN queries and A_{TS} makes q_s SIGN queries. The running times of $A_{\mathsf{S}}, A_{\mathsf{TS}}$ are close to that of A.

EXTENSIONS AND APPLICATIONS. Like hashing, signature verification has applications in certificate-based authentication, which we discuss in the following section. In contrast to hashing and NIAs, signatures have per-user keys, which raises a multi-user question. We have presented single-user definitions in this section but one can consider multi-user uf-cma and ffe. Our results extend to this case, notably because the exploiter-chosen vk' is included in the signature produced by the exploit algorithm. Therefore an exploit for a particular vk'_1 cannot be reused for a different vk'_2.

8 PA-SAs applied to certificate forgery

In this section we discuss certificate forgery as a possible target of PA-SAs. Both hash functions and signature verification, as covered in Sections 5 and 7, are components of certificates as used in public-key infrastructure. Do PA-SAs in our predicate-based formalization allow for realistic certificate attacks? Yes; we explain how in this section.

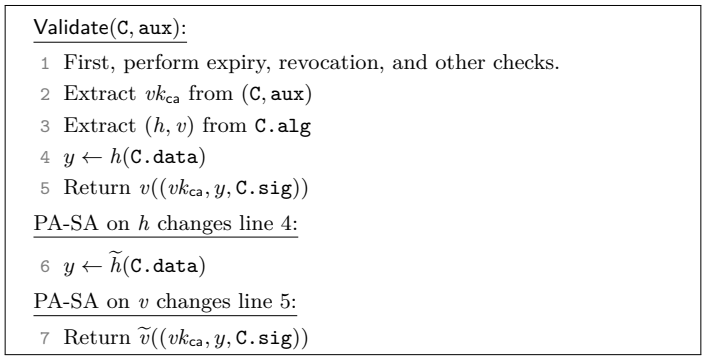

Figure 13: Validation of an X.509 certificate, simplified. Recall that $v(\cdot) = \mathsf{S_{ca}}.\mathsf{Vfy}(\cdot)$.

CERTIFICATE-BASED AUTHENTICATION. First let us recall how certificates are used. The usual realization of public-key infrastructure uses X.509 certificates and certificate authorities. For simplicity, suppose there is one CA operating with signature scheme $\mathsf{S_{ca}}$ and hash function $h \in \mathsf{H_{ca}}$. Let $(vk_{\mathsf{ca}}, sk_{\mathsf{ca}})$ be the verification and signing keys of the CA; in our PA-SA syntax the signature verification algorithm is $v = \mathsf{S_{ca}}.\mathsf{Vfy}$. All users of the PKI have implementations of h and v (along with vk_{ca}) on their devices.

As specified in RFC 5280 [24], a certificate C consists of a sequence of key-value pairs. The important fields for our application are:

- C.data, consisting of the certificate's identifying, validity and other data. At a minimum, this specifies the CA who signed the certificate and includes information to recover vk_{ca}.
- C.alg, the name of the signature algorithm, such as "PKCS #1 SHA-256 With RSA Encryption." This yields algorithms h and v.
- C.sig, a signature on message C.data, using the algorithm specified in C.alg and the CA's signing key sk_{ca}.

Issuance of a certificate takes as input data$'$ and auxiliary information csr (representing a certificate signing request) to produce either $\perp$, or a signed certificate C containing data$'$. Deterministic validation of a certificate takes as input a certificate C and auxiliary information aux (representing a certificate chain, and local store of root certificates) to produce a bit $d \in \{0, 1\}$.

In our discussion, we are more interested in validation (the public algorithm) than issuance (the secretly-keyed algorithm). At a high level, Validate proceeds as in Figure 13. Validate itself can be treated as a public algorithm to which our general framework applies, but we here discuss PA-SAs on the hash function h or signature verification v.

PA-SA ON SIGNATURE VERIFICATION. We first discuss the simpler case, when signature verification is attacked. Let $(\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{S_{ca}}}(v)$ for a PA-SA $\widetilde{\mathsf{S_{ca}}}$ on the CA's signature scheme. As in Figure 13, an attacker substitutes v with $\widetilde{v}$ on a user's device. Recall that signature PA-SA utility means that for any verification key vk and message m, one can use $e(u = (vk, m), 1)$ to find a signature σ under which $\widetilde{v}((vk, m, \sigma)) = 1$. For impersonation one selects $vk = vk_{\mathsf{ca}}$.

To mount a certificate attack, one chooses arbitrary certificate data data* and computes $m^* = h($data$^*)$. Then the exploit algorithm allows computation of sig$^* \leftarrow e((vk_{\mathsf{ca}}, m^*), 1)$ such that $\widetilde{v}((vk_{\mathsf{ca}}, m^*,$ sig$^*)) = 1$. The full certificate C* may be put together with the chosen data*, alg* specifying the usual h and v, and signature sig*. Now line 7 of Figure 13 passes and the certificate is accepted. This enables, for example, impersonation by choosing any vk_{ca} to claim in data*. One needs only to instantiate $\widetilde{\mathsf{S_{ca}}} = \mathbf{SbvIt}[\mathsf{S_{ca}}, \mathsf{S}, \mathsf{Emb_{ff}}]$ with a signature scheme such that $\mathsf{S.sl} \leq \mathsf{S_{ca}.sl}$

$\mathsf{P}_{\mathsf{cert}}(x = \mathtt{data}^*, u = \mathtt{data}')$:

1 If $((\mathtt{data}^*.\mathtt{f} = \mathtt{data}'.\mathtt{f})$ for all fields $\mathtt{f}$ except $\mathtt{f}_h, \mathtt{f}_\sigma)$ then return true
2 Else return false

$\mathsf{Emb}_{\mathsf{cert}}((y \,\|\, \sigma), u = \mathtt{data}')$:

3 For all fields $\mathtt{f}$, do: $\mathtt{data}^*.\mathtt{f} \leftarrow \mathtt{data}'.\mathtt{f}$
4 $\mathtt{data}^*.\mathtt{f}_h \leftarrow y$; $\mathtt{data}^*.\mathtt{f}_\sigma \leftarrow \sigma$
5 Return $\mathtt{data}^*$

$\mathsf{EmbInv}_{\mathsf{cert}}(x = \mathtt{data}^*)$:

6 $y \leftarrow \mathtt{data}^*.\mathtt{f}_h$; $\sigma \leftarrow \mathtt{data}^*.\mathtt{f}_\sigma$
7 $\mathtt{data}'.\mathtt{f} \leftarrow \mathtt{data}^*.\mathtt{f}$ for all fields $\mathtt{f}$ except $\mathtt{f}_h, \mathtt{f}_\sigma$
8 Return $((y \,\|\, \sigma), \mathtt{data}')$

Certificate forgery using $\widetilde{\mathsf{H}_{\mathsf{ca}}}$:

9 $(\tilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}_{\mathsf{ca}}}(h)$
10 Choose a target honest certificate $\mathtt{C}$ with hash y.
11 Choose any $\mathtt{data}'$ which specifies the same issuer CA and vk_{ca} as $\mathtt{C}$.
12 $\mathtt{data}^* \leftarrow\!\!\$\ e(u = \mathtt{data}', y)$
13 $\mathtt{C}^*.\mathtt{data} \leftarrow \mathtt{data}^*$
14 $\mathtt{C}^*.\mathtt{alg}$ specifies (h, v)
15 $\mathtt{C}^*.\mathtt{sig} \leftarrow \mathtt{C}.\mathtt{sig}$
16 Return $\mathtt{C}^*$

Figure 14: **Above:** Predicate $\mathsf{P}_{\mathsf{cert}}$ which captures whether $\mathtt{data}^*$ embeds all of the data from chosen $\mathtt{data}'$. Embedding $\mathsf{Emb}_{\mathsf{cert}}$ is a embedding function for this predicate. **Below:** How a PA-SA on hash function h allows the attacker to forge almost arbitrary certificates. We use $\mathtt{f}$ to denote any X.509 data field name. Field labels $\mathtt{f}_h, \mathtt{f}_\sigma$ are fixed.

---

so that the results of Section 7 apply.

PA-SA ON THE HASH FUNCTION. Next we discuss a PA-SA on the hash function, for which the main task is deciding on an embedding to use. Recall that utility of a PA-SA on a hash function is with respect to a particular predicate, and our construction assumes a compatible embedding function for that predicate. We will call these $\mathsf{P}_{\mathsf{cert}}$ and $\mathsf{Emb}_{\mathsf{cert}}$, respectively, and define them shortly. For now, let $\widetilde{\mathsf{H}_{\mathsf{ca}}} = \mathbf{SbvIt}[\mathsf{H}_{\mathsf{ca}}, \mathsf{S}, \mathsf{Emb}_{\mathsf{cert}}]$ given a signature scheme S with *short* signatures. Let $(\tilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}_{\mathsf{ca}}}(h)$. As in Figure 13, an attacker substitutes h with $\tilde{h}$ on a user's device.

Now, certificate forgery proceeds by choosing a target certificate $\mathtt{C}$ with hash y for which a CA signature is already known; thus the signature may be reused on any $\mathtt{C}^*$ whose data also hashes to y. Unlike for the signature case above, we cannot ask for arbitrary certificate data to hash to y. However, we can ask that for arbitrary $\mathtt{data}'$, the attacker can find $\mathtt{data}^*$ which hashes to y and is close to $\mathtt{data}'$. This "close to" is captured by the predicate and embedding function. We give one possible $\mathsf{P}_{\mathsf{cert}}$ and $\mathsf{Emb}_{\mathsf{cert}}$ in Figure 14. In brief, $\mathtt{data}^*$ contains all the same fields as $\mathtt{data}'$, and two additional fields $\mathtt{f}_h, \mathtt{f}_\sigma$. One may also design other ways to embed $(\mathsf{H}_{\mathsf{ca}}.\mathsf{ol} + \mathsf{S}.\mathsf{sl})$ bits in a certificate. The steps of the forgery are written in more detail in Figure 14 and we give an explicit example in Figure 15.

REMARKS ON OTHER PKI ATTACKS. Although we consider the PA-SA setting, we note that a variety of work has already studied the repercussions of PKI allowing weak hash functions, in particular MD5 [49,50,64,68,72]. At Eurocrypt 2007 [63], Stevens, Lenstra, and de Weger presented two X.509 certificates with the same MD5 hash value and thus signature. The two certificates were produced at the same time, and the cost was estimated to be "2 months real time" [63]. In the

PA-SA model, an attacker wants to forge (almost) arbitrary certificates, at arbitrary times, and to do so easily.

There are a few remarks to add about our PA-SAs here. First, observe that the hash function exploit operates as $\texttt{data}^* \leftarrow\!\!\$\ e(u = \texttt{data}', y)$ where there is some restriction on the vk_{ca} in $\texttt{data}'$ (there must be an existing public certificate and signature under vk_{ca}). The signature exploit operates as $\texttt{sig}^* \leftarrow e(u = (vk_{\mathsf{ca}}, m^*), 1)$ where vk_{ca} is totally arbitrary. Second, in both, vk_{ca} is included in the constraint-parameter u, meaning that an exploit-produced signature depends on vk_{ca} and could not be reused for a different vk'_{ca}.

Example certificate embedding. In Figure 15 we give an explicit example of our embedding $\mathsf{Emb}_{\mathsf{cert}}$, as used in a PA-SA on the PKI hash function. Suppose we have selected target hash $y =$ `680f8b1123be39f4451430d6267a8159033034403ce0df1abdf11c105031d719`. This corresponds to a public certificate `C` with a valid signature `C.sig` where `C.alg` specifies "PKCS #1 SHA-256 With RSA Encryption." The aim is now to construct $\texttt{C}^* \neq \texttt{C}$ with the same hash y; thus the signature on `C` can be reused.

A PA-SA attacker does the following. Suppose they intend to use $\widetilde{\mathsf{H}_{\mathsf{ca}}} = \mathbf{SbvIt}[\mathsf{H}_{\mathsf{ca}}, \mathsf{S}, \mathsf{Emb}_{\mathsf{cert}}]$ where H_{ca} is $\mathsf{SHA256}$, S is ECDSA over `secp256k1`, and the embedding is as in Figure 14. (A shorter signature scheme like BLS [17] may be even less noticeable to embed, but ECDSA is easily used in OpenSSL [53].)

The forgery proceeds as follows. First, the substitution is generated via $(\tilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}_{\mathsf{ca}}}(h)$, which in particular means generating $(\widetilde{vk}, \widetilde{sk})$ for ECDSA. (Recall that $\widetilde{vk}$ is hardcoded in $\tilde{h}$ while $\widetilde{sk}$ is hardcoded in e.) We select:

$$\widetilde{vk} = \texttt{04d0722759460447f1719ac66a1734054651f7c557a96166583d686}$$
$$\texttt{ad405ca9b6f5fe47a7e425a8722edfa13be606fcbe4053ecacb27f2}$$
$$\texttt{b0bc3dd1e83152c9a8a3}\ .$$

Next $\texttt{data}'$ is chosen, which is the certificate data to be contained in the forgery. For this example, we suppose that the attacker is aiming to forge arbitrary $\texttt{data}'$. They use e to find a preimage $\texttt{data}^*$ of target hash y, where the constraint is that $\texttt{data}^*$ is "close to" $\texttt{data}'$. Concretely, $\texttt{data}^*$ adds two additional fields. In the first, $\texttt{data}^*.\texttt{f}_h = y$. In the second, $\texttt{data}^*.\texttt{f}_\sigma = \sigma$, where $\sigma \leftarrow\!\!\$\ \mathsf{S.Sign}(\widetilde{sk}, (y, \texttt{data}'))$. For our chosen data and y, we find

$$\sigma = \texttt{304502202b978f95a853dfa2d2574ff9980a4351e7d6c9c4fcc0529}$$
$$\texttt{d636c750fdf4c16a8022100efbb50c105df2a4766cfa94910d3a190}$$
$$\texttt{19656ff5dbdeed8948eb7570089e12d5}\ .$$

Now the forgery is ready to be put together: it includes $\texttt{data}^*$, signature `C.sig`, and algorithm specification "PKCS #1 SHA-256 With RSA Encryption" (where $h = \mathsf{SHA256}$ is substituted by $\tilde{h}$ on the user's device.) This is shown in Figure 15.

Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions.

Certificate $\texttt{C}^*$:

```
Data:
  Version: 3 (0x2)
  Serial Number: ...
  Signature Algorithm: PKCS #1 SHA-256 With RSA Encryption
  Issuer: C.Issuer
  Validity
    Not Before: Jan 1 08:00:00 2024 GMT
    Not After: Dec 1 08:00:00 2024 GMT
  Subject: O = Big Brother, CN = *.bigbrother.com
  Subject Public Key Info: ...
  X509v3 extensions:
    X509v3 Basic Constraints: critical
      CA: TRUE
    f_h: 680f8b1123be39f4451430d6267a8159033034403ce0df1abdf11c105031d719
    f_σ: 304502202b978f95a853dfa2d2574ff9...56ff5dbdeed8948eb7570089e12d5
Signature Algorithm: PKCS #1 SHA-256 With RSA Encryption
C.sig
```

Figure 15: A certificate forgery for a PA-SA on h; we want $\tilde{h}(\texttt{C}^*.\texttt{data}) = \tilde{h}(\texttt{C}.\texttt{data}) = h(\texttt{C}.\texttt{data})$ so that an honest signature on $\texttt{C}$ can be reused for forgery $\texttt{C}^*$. The highlighted lines are the overhead in constructing the hash collision; that is, these are determined by $\texttt{C}$ or cannot be arbitrarily chosen. The remainder of the certificate may be arbitrarily set by the PA-SA attacker. On a user's device, h is replaced by $\tilde{h}$.

References

[1] H. Abelson, R. Anderson, S. M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, M. Green, S. Landau, P. G. Neumann, R. L. Rivest, J. I. Schiller, B. Schneier, M. Specter, and D. J. Weitzner. Keys under doormats: Mandating insecurity by requiring government access to all data and communications, 2015. https://dspace.mit.edu/bitstream/handle/1721.1/97690/MIT-CSAIL-TR-2015-026.pdf. (Cited on 5.)

[2] A. Albertini, J.-P. Aumasson, M. Eichlseder, F. Mendel, and M. Schläffer. Malicious hashing: Eve's variant of SHA-1. In A. Joux and A. M. Youssef, editors, *SAC 2014*, volume 8781 of *LNCS*, pages 1–19. Springer, Cham, Aug. 2014. (Cited on 4, 6, 14.)

[3] M. Armour and B. Poettering. Subverting decryption in AEAD. In M. Albrecht, editor, *17th IMA International Conference on Cryptography and Coding*, volume 11929 of *LNCS*, pages 22–41. Springer, Cham, Dec. 2019. (Cited on 2, 5.)

[4] G. Ateniese, D. Francati, B. Magri, and D. Venturi. Public immunization against complete subversion without random oracles. In R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, editors, *ACNS 19International Conference on Applied Cryptography and Network Security*, volume 11464 of *LNCS*, pages 465–485. Springer, Cham, June 2019. (Cited on 5.)

[5] G. Ateniese, B. Magri, and D. Venturi. Subversion-resilient signature schemes. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 364–375. ACM Press, Oct. 2015. (Cited on 2, 5, 19.)

[6] B. Auerbach, M. Bellare, and E. Kiltz. Public-key encryption resistant to parameter subversion and its realization from efficiently-embeddable groups. In M. Abdalla and R. Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 348–377. Springer, Cham, Mar. 2018. (Cited on 6.)

[7] F. Banfi, K. Gegier, M. Hirt, U. Maurer, and G. Rito. Anamorphic encryption, revisited. In M. Joye and G. Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 3–32. Springer, Cham, May 2024. (Cited on 6.)

[8] M. Bellare, G. Fuchsbauer, and A. Scafuro. NIZKs with an untrusted CRS: Security in the face of parameter subversion. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 777–804. Springer, Berlin, Heidelberg, Dec. 2016. (Cited on 5, 6, 17.)

[9] M. Bellare, J. Jaeger, and D. Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1431–1440. ACM Press, Oct. 2015. (Cited on 2, 5, 12.)

[10] M. Bellare, K. G. Paterson, and P. Rogaway. Security of symmetric encryption against mass surveillance. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Berlin, Heidelberg, Aug. 2014. (Cited on 2, 3, 5, 9.)

[11] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006. (Cited on 6, 10, 14, 32.)

[12] P. Bemmann, S. Berndt, and R. Chen. Subversion-resilient signatures without random oracles. In C. Pöpper and L. Batina, editors, *Applied Cryptography and Network Security*, pages 351–375, Cham, 2024. Springer Nature Switzerland. (Cited on 14.)

[13] P. Bemmann, R. Chen, and T. Jager. Subversion-resilient public key encryption with practical watchdogs. In J. Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 627–658. Springer, Cham, May 2021. (Cited on 5.)

[14] S. Berndt, J. Wichelmann, C. Pott, T.-H. Traving, and T. Eisenbarth. ASAP: Algorithm substitution attacks on cryptographic protocols. In Y. Suga, K. Sakurai, X. Ding, and K. Sako, editors, *ASIACCS 22*, pages 712–726. ACM Press, May / June 2022. (Cited on 2, 5.)

[15] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. (Cited on 5.)

[16] D. J. Bernstein, T. Lange, and R. Niederhagen. Dual EC: A standardized back door. In P. Ryan, D. Naccache, and J.-J. Quisquater, editors, *The New Codebreakers*. Springer, Heidelberg, 2016. (Cited on 5.)

[17] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Berlin, Heidelberg, Dec. 2001. (Cited on 25.)

[18] D. Boneh, E. Shen, and B. Waters. Strongly unforgeable signatures based on computational Diffie-Hellman. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 229–240. Springer, Berlin, Heidelberg, Apr. 2006. (Cited on 7.)

[19] S. Chakraborty, C. Ganesh, and P. Sarkar. Reverse firewalls for oblivious transfer extension and applications to zero-knowledge. In C. Hazay and M. Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 239–270. Springer, Cham, Apr. 2023. (Cited on 17.)

[20] R. Chen, X. Huang, and M. Yung. Subvert KEM to break DEM: Practical algorithm-substitution attacks on public-key encryption. In S. Moriai and H. Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 98–128. Springer, Cham, Dec. 2020. (Cited on 2, 5.)

[21] S. S. M. Chow, A. Russell, Q. Tang, M. Yung, Y. Zhao, and H.-S. Zhou. Let a non-barking watchdog bite: Cliptographic signatures with an offline watchdog. In D. Lin and K. Sako, editors, *PKC 2019, Part I*, volume 11442 of *LNCS*, pages 221–251. Springer, Cham, Apr. 2019. (Cited on 2.)

[22] B. Cogliati, J. Ethan, and A. Jha. Subverting telegram's end-to-end encryption. IACR *Transactions on Symmetric Cryptology*, 2023. https://tosc.iacr.org/index.php/ToSC/article/view/10302/9747. (Cited on 2, 5.)

[23] S. Contini, A. K. Lenstra, and R. Steinfeld. VSH, an efficient and provable collision-resistant hash function. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 165–182. Springer, Berlin, Heidelberg, May / June 2006. (Cited on 4, 14.)

[24] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, May 2008. (Cited on 23.)

[25] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. In J. Motiwalla and G. Tsudik, editors, *ACM CCS 99*, pages 46–51. ACM Press, Nov. 1999. (Cited on 7.)

[26] C. Crépeau and A. Slakmon. Simple backdoors for RSA key generation. In M. Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 403–416. Springer, Berlin, Heidelberg, Apr. 2003. (Cited on 5, 19.)

[27] J. P. Degabriele, P. Farshim, and B. Poettering. A more cautious approach to security against mass surveillance. In G. Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 579–598. Springer, Berlin, Heidelberg, Mar. 2015. (Cited on 2, 5.)

[28] J. P. Degabriele, K. G. Paterson, J. C. N. Schuldt, and J. Woodage. Backdoors in pseudorandom number generators: Possibility and impossibility results. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 403–432. Springer, Berlin, Heidelberg, Aug. 2016. (Cited on 6.)

[29] D. Derler, K. Samelin, and D. Slamanig. Bringing order to chaos: The case of collision-resistant chameleon-hashes. In A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 462–492. Springer, Cham, May 2020. (Cited on 14.)

[30] Y. Dodis, C. Ganesh, A. Golovnev, A. Juels, and T. Ristenpart. A formal treatment of backdoored pseudorandom generators. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 101–126. Springer, Berlin, Heidelberg, Apr. 2015. (Cited on 6.)

[31] Y. Dodis, I. Mironov, and N. Stephens-Davidowitz. Message transmission with reverse firewalls—secure communication on corrupted machines. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 341–372. Springer, Berlin, Heidelberg, Aug. 2016. (Cited on 5.)

[32] N. Döttling, S. Garg, Y. Ishai, G. Malavolta, T. Mour, and R. Ostrovsky. Trapdoor hash functions and their applications. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 3–32. Springer, Cham, Aug. 2019. (Cited on 14.)

[33] E. Felten. The Linux backdoor attempt of 2003, 2013. https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/. (Cited on 5.)

[34] M. Fischlin and F. Günther. Verifiable verification in cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3239–3253, New York, NY, USA, 2023. Association for Computing Machinery. (Cited on 17.)

[35] M. Fischlin, C. Janson, and S. Mazaheri. Backdoored hash functions: Immunizing HMAC and HKDF. In S. Chong and S. Delaune, editors, *CSF 2018 Computer Security Foundations Symposium*, pages 105–118. IEEE Computer Society Press, 2018. (Cited on 2, 4, 6, 11, 14, 15.)

[36] A. Freund. Openwall oss-security mailing list, 29 Mar. 2024. https://www.openwall.com/lists/oss-security/2024/03/29/4. (Cited on 6.)

[37] G. Fuchsbauer. Subversion-zero-knowledge SNARKs. In M. Abdalla and R. Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 315–347. Springer, Cham, Mar. 2018. (Cited on 5, 17.)

[38] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 123–139. Springer, Berlin, Heidelberg, May 1999. (Cited on 7.)

[39] E.-J. Goh, D. Boneh, B. Pinkas, and P. Golle. The design and implementation of protocol-based hidden key recovery. In C. Boyd and W. Mao, editors, *ISC 2003*, volume 2851 of *LNCS*, pages 165–179. Springer, Berlin, Heidelberg, Oct. 2003. (Cited on 2, 5.)

[40] O. Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004. (Cited on 7.)

[41] S. Goldwasser, M. P. Kim, V. Vaikuntanathan, and O. Zamir. Planting undetectable backdoors in machine learning models : [extended abstract]. In *63rd FOCS*, pages 931–942. IEEE Computer Society Press, Oct. / Nov. 2022. (Cited on 2, 3, 6, 12.)

[42] P. Hodges and D. Stebila. Algorithm substitution attacks: State reset detection and asymmetric modifications. *IACR Trans. Symm. Cryptol.*, 2021(2):389–422, 2021. (Cited on 2.)

[43] D. Hofheinz and E. Kiltz. Programmable hash functions and their applications. *Journal of Cryptology*, 25(3):484–527, July 2012. (Cited on 14.)

[44] T. Horel, S. Park, S. Richelson, and V. Vaikuntanathan. How to subvert backdoored encryption: Security against adversaries that decrypt all ciphertexts. In A. Blum, editor, *ITCS 2019*, volume 124, pages 42:1–42:20. LIPIcs, Jan. 2019. (Cited on 6.)

[45] A. Joux, J. Loss, and B. Wagner. Kleptographic attacks against implicit rejection. Cryptology ePrint Archive, Report 2024/260, 2024. https://eprint.iacr.org/2024/260. (Cited on 2, 5.)

[46] B. Kaliski. PKCS #5: Password-based cryptography specification. RSA Laboratories, Sept. 2000. Version 2.0. (Cited on 16.)

[47] H. Krawczyk and T. Rabin. Chameleon signatures. In *NDSS 2000*. The Internet Society, Feb. 2000. (Cited on 14.)

[48] M. Kutylowski, G. Persiano, D. H. Phan, M. Yung, and M. Zawada. Anamorphic signatures: Secrecy from a dictator who only permits authentication! In H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 759–790. Springer, Cham, Aug. 2023. (Cited on 6.)

[49] A. Lenstra, X. Wang, and B. de Weger. Colliding x.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005. (Cited on 24.)

[50] A. K. Lenstra and B. de Weger. On the possibility of constructing meaningful hash collisions for public keys. In C. Boyd and J. M. G. Nieto, editors, *ACISP 05*, volume 3574 of *LNCS*, pages 267–279. Springer, Berlin, Heidelberg, July 2005. (Cited on 24.)

[51] I. Mironov and N. Stephens-Davidowitz. Cryptographic reverse firewalls. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 657–686. Springer, Berlin, Heidelberg, Apr. 2015. (Cited on 5.)

[52] National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-5, Oct. 2019. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5-draft.pdf. (Cited on 5.)

[53] OpenSSL. https://www.openssl.org/. (Cited on 25.)

[54] G. Persiano, D. H. Phan, and M. Yung. Anamorphic encryption: Private communication against a dictator. In O. Dunkelman and S. Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 34–63. Springer, Cham, May / June 2022. (Cited on 6.)

[55] T. Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). RFC 6979, Aug. 2013. https://datatracker.ietf.org/doc/html/rfc6979. (Cited on 5.)

[56] P. Ravi, S. Bhasin, A. Chattopadhyay, Aikata, and S. S. Roy. Backdooring post-quantum cryptography: Kleptographic attacks on lattice-based KEMs. Cryptology ePrint Archive, Report 2022/1681, 2022. (Cited on 2, 5.)

[57] A. Russell, Q. Tang, M. Yung, and H.-S. Zhou. Cliptography: Clipping the power of kleptographic attacks. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 34–64. Springer, Berlin, Heidelberg, Dec. 2016. (Cited on 2, 5.)

[58] A. Russell, Q. Tang, M. Yung, and H.-S. Zhou. Generic semantic security against a kleptographic adversary. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 907–922. ACM Press, Oct. / Nov. 2017. (Cited on 2, 5.)

[59] A. Russell, Q. Tang, M. Yung, and H.-S. Zhou. Correcting subverted random oracles. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 241–271. Springer, Cham, Aug. 2018. (Cited on 14.)

[60] D. Shumow and N. Ferguson. On the possibility of a back door in the NIST SP800-90 Dual EC PRNG. *CRYPTO'07* Rump Session, 2007. https://rump2007.cr.yp.to/15-shumow.pdf. (Cited on 5.)

[61] G. J. Simmons. The prisoners' problem and the subliminal channel. In D. Chaum, editor, *CRYPTO'83*, pages 51–67. Plenum Press, New York, USA, 1983. (Cited on 6.)

[62] G. J. Simmons. The subliminal channel and digital signature. In T. Beth, N. Cot, and I. Ingemarsson, editors, *EUROCRYPT'84*, volume 209 of *LNCS*, pages 364–378. Springer, Berlin, Heidelberg, Apr. 1985. (Cited on 6.)

[63] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In M. Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 1–22. Springer, Berlin, Heidelberg, May 2007. (Cited on 24.)

[64] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, Berlin, Heidelberg, Aug. 2009. (Cited on 24.)

[65] T. Tiemann, S. Berndt, T. Eisenbarth, and M. Liskiewicz. "Act natural!": Having a private chat on a public blockchain. Cryptology ePrint Archive, Report 2021/1073, 2021. (Cited on 2, 5, 19.)

[66] P. R. Tiwari and M. Green. Subverting cryptographic hardware used in blockchain consensus. In J. Clark and E. Shi, editors, *FC 2024*, 2024. (Cited on 2, 5, 17.)

[67] F. Valsorda. Bluesky post, 30 Mar. 2024. https://bsky.app/profile/did:plc:x2nsupeeo52oznrmplwapppl/post/3kowjkx2njy2b. (Cited on 6.)

[68] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 19–35. Springer, Berlin, Heidelberg, May 2005. (Cited on 24.)

[69] A. Young and M. Yung. The dark side of "black-box" cryptography, or: Should we trust capstone? In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 89–103. Springer, Berlin, Heidelberg, Aug. 1996. (Cited on 2, 5.)

[70] A. Young and M. Yung. Kleptography: Using cryptography against cryptography. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 62–74. Springer, Berlin, Heidelberg, May 1997. (Cited on 2, 5.)

[71] A. Young and M. Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In B. S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 264–276. Springer, Berlin, Heidelberg, Aug. 1997. (Cited on 5.)

[72] G. Zaverucha and D. Shumow. Are certificate thumbprints unique? Cryptology ePrint Archive, Report 2019/130, 2019. (Cited on 24.)

Games G_0, $\boxed{\mathrm{G}_1}$, $\boxed{\mathrm{G}_2}$

INIT:
1 $h \leftarrow\!\!\$\ \mathsf{H}$; $(\widetilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}}(h)$; $\mathcal{X} \leftarrow \emptyset$
2 Return $(h, \widetilde{h})$

GETPMG(u, y):
3 $x \leftarrow\!\!\$\ e(u, y)$; $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$
4 Return x

FIN(x_1, x_2):
5 If $(x_1 = x_2)$ then return false
6 If $(x_1 \in \mathcal{X}) \vee (x_2 \in \mathcal{X})$ then return false
7 If $\widetilde{h}(x_1) \neq h(x_1) \vee \widetilde{h}(x_2) \neq h(x_2)$ then bad $\leftarrow$ true ; $\boxed{\text{return false}}$
8 $y_1 \leftarrow \widetilde{h}(x_1)$; $y_2 \leftarrow \widetilde{h}(x_2)$
9 $y_1 \leftarrow h(x_1)$; $y_2 \leftarrow h(x_2)$ // Game G_2
10 Return $(y_1 = y_2)$

Figure 16: Games $\mathrm{G}_0, \mathrm{G}_1, \mathrm{G}_2$ for the proof of Theorem 5.1. $\mathrm{G}_1, \mathrm{G}_2$ contain the boxed code and G_0 does not. Line 9 is only present in G_2.

Adversary $A_{\widetilde{\mathsf{H}}}(h, \widetilde{h}, \emptyset)$:
1 $\mathcal{X} \leftarrow \emptyset$
2 $(x_1, x_2) \leftarrow A[\mathrm{GETPMG}_{\widetilde{\mathsf{H}}}](h, \widetilde{h})$
3 If $(x_1 = x_2)$ then return $\perp$
4 If $(x_1 \in \mathcal{X}) \vee (x_2 \in \mathcal{X})$ then return $\perp$
5 If $\widetilde{h}(x_1) \neq h(x_1)$ then return x_1
6 If $\widetilde{h}(x_2) \neq h(x_2)$ then return x_2

Oracle $\mathrm{GETPMG}_{\widetilde{\mathsf{H}}}(u, y)$:
7 $x \leftarrow \mathrm{GETPMG}((y, u))$
8 $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$; Return x

Adversary $A_{\mathsf{H}}(h)$:
1 $(\widetilde{h}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{H}}(h)$; $\mathcal{X} \leftarrow \emptyset$
2 $(x_1, x_2) \leftarrow A[\mathrm{GETPMG}_{\mathsf{H}}](h, \widetilde{h})$
3 Return (x_1, x_2)

Oracle $\mathrm{GETPMG}_{\mathsf{H}}(u, y)$:
4 $x \leftarrow\!\!\$\ e(u, y)$; $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$
5 Return x

Figure 17: Adversaries $A_{\widetilde{\mathsf{H}}}$ (left) and A_{H} (right) for the proof of Theorem 5.1.

A Proof of Theorem 5.1

Proof of Theorem 5.1: Consider game G_0 of Figure 16. It is easy to see that this game is exactly the cfe game. We already include line 7 which sets a flag bad, but in G_0 this has no effect on the final output. Hence,

$$\mathbf{Adv}^{\mathrm{cfe}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A) = \Pr[\,\mathrm{G}_0(A)\,] .$$

We next turn to game G_1 which outputs false whenever bad is set. Therefore, games $\mathrm{G}_0, \mathrm{G}_1$ are identical-until-bad and by the Fundamental Lemma of Game Playing [11] we have

$$\begin{aligned}\Pr[\,\mathrm{G}_0(A)\,] &= \Pr[\,\mathrm{G}_1(A)\,] + (\Pr[\,\mathrm{G}_0(A)\,] - \Pr[\,\mathrm{G}_1(A)\,]) \\ &\leq \Pr[\,\mathrm{G}_1(A)\,] + \Pr[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,] .\end{aligned}$$

Let us now take a closer look at the event defined in line 7. The game outputs false when the output of $\widetilde{h}$ and h differs on either x_1 or x_2 which is exactly captured by the exclusivity of $\widetilde{\mathsf{H}}$. That

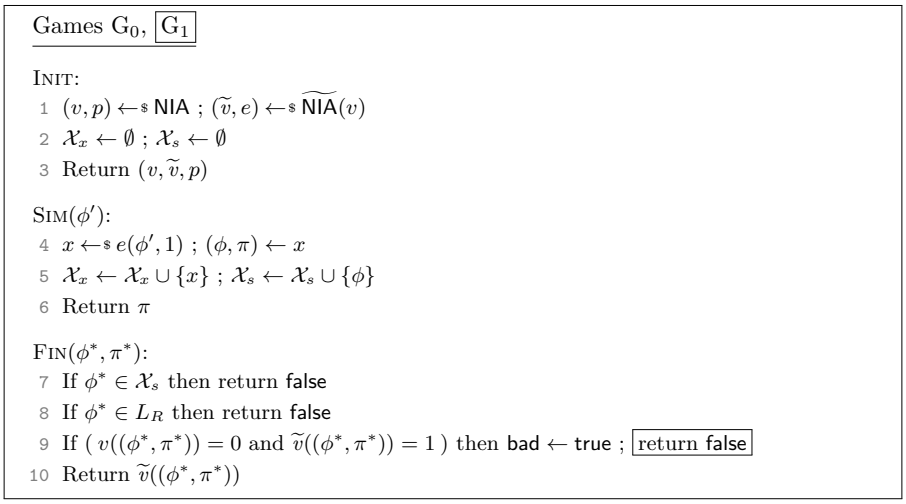

Figure 18: Games G_0, G_1 for the proof of Theorem 6.1. G_1 contains the boxed code and G_0 does not.

---

is, we can construct an adversary $A_{\widetilde{\mathsf{H}}}$ for which

$$\Pr[\,G_1(A) \text{ sets } \mathsf{bad}\,] \le \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}(A_{\widetilde{\mathsf{H}}})\ . \tag{11}$$

Adversary $A_{\widetilde{\mathsf{H}}}$ is in game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{H},\widetilde{\mathsf{H}},\mathsf{P}}$ and runs A as specified on the left side of Figure 17. $A_{\widetilde{\mathsf{H}}}$ simulates A's preimage oracle using its own preimage oracle. Suppose now that $\mathsf{bad} \leftarrow \mathsf{true}$ on line 7 of G_1. Then A has output x_1, x_2 such that $\widetilde{h}(x_i) \neq h(x_i)$ for at least one $i \in \{1,2\}$, while both are not in $\mathcal{X}$, meaning have not been the output of a query to GETPMG. Hence, $A_{\widetilde{\mathsf{H}}}$ has found a winning output $x^* \in \{x_1, x_2\}$ in the exclusivity game. This proves Eq. (11).

We next turn to game G_2, where the assignment in line 8 is replaced by the one in line 9. More specifically, y_1 and y_2 are now computed using h instead of using $\widetilde{h}$. We claim that

$$\Pr[\,G_2(A)\,] = \Pr[\,G_1(A)\,]\ . \tag{12}$$

To justify Eq. (12), we observe that whenever the outputs of h and $\widetilde{h}$ for inputs x_1 or x_2 differ, the game has already returned false. Hence, we have $h(x_i) = \widetilde{h}(x_i)$ for both $i \in \{1,2\}$.

Finally, we claim that

$$\Pr[\,G_2(A)\,] \le \mathbf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(A_{\mathsf{H}})\ . \tag{13}$$

We construct adversary A_{H} in game $\mathbf{G}^{\mathrm{cr}}_{\mathsf{H}}$ as specified on the right side of Figure 17. A's view is that of game G_2; initialization and $\textsc{GetPmg}_{\mathsf{H}}$ return the same responses as in G_2. Now, if $G_2(A)$ returns true, and since the boxed code is executed in G_2, then it must be that $y_1 = y_2$ and thus $h(x_1) = h(x_2)$. This is precisely the winning condition of A_{H}'s game $\mathbf{G}^{\mathrm{cr}}_{\mathsf{H}}$ and proves Eq. (13). A_{H} maintains running time close to that of A. ∎

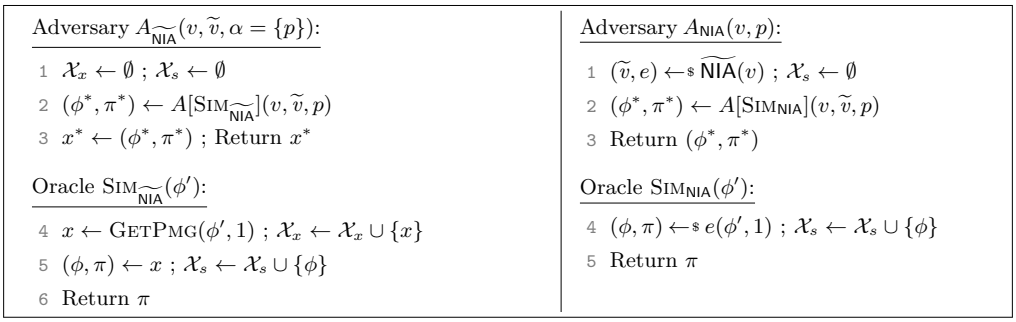

Figure 19: Adversaries $A_{\widetilde{\mathsf{NIA}}}$ (left) and A_{NIA} (right) for the proof of Theorem 6.1.

B Proof of Theorem 6.1

Proof of Theorem 6.1: Consider game G_0 of Figure 18. We rewrite the winning condition by splitting it into three checks. We already include line 9 which sets a flag bad. We have

$$\mathbf{Adv}^{\mathrm{pfe}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}}}(A) = \Pr\left[\,\mathrm{G}_0(A)\,\right].$$

We next turn to game G_1 which outputs false whenever bad is set; that is, when $\widetilde{v}((\phi^*, \pi^*))$ is true, but $v((\phi^*, \pi^*))$ is not. Therefore, games $\mathrm{G}_0, \mathrm{G}_1$ are identical-until-bad and we have

$$\begin{aligned}\Pr\left[\,\mathrm{G}_0(A)\,\right] &= \Pr\left[\,\mathrm{G}_1(A)\,\right] + \left(\Pr\left[\,\mathrm{G}_0(A)\,\right] - \Pr\left[\,\mathrm{G}_1(A)\,\right]\right)\\ &\le \Pr\left[\,\mathrm{G}_1(A)\,\right] + \Pr\left[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,\right].\end{aligned}$$

We now construct adversaries $A_{\widetilde{\mathsf{NIA}}}, A_{\mathsf{NIA}}$ such that the following two equations hold:

$$\Pr\left[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,\right] \le \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}},\mathsf{P}_{\mathsf{pf}}}(A_{\widetilde{\mathsf{NIA}}}) \tag{14}$$

$$\Pr\left[\,\mathrm{G}_1(A)\,\right] \le \mathbf{Adv}^{\mathrm{snd}}_{\mathsf{NIA}}(A_{\mathsf{NIA}}), \tag{15}$$

which will complete the proof of Eq. (7) in the theorem statement.

We begin with $A_{\widetilde{\mathsf{NIA}}}$ which is in game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}},\mathsf{P}_{\mathsf{pf}}}$ and operates according to the description in Figure 19. $A_{\widetilde{\mathsf{NIA}}}$ runs A, responding to oracle queries via $\text{SIM}_{\widetilde{\mathsf{NIA}}}$. These follow the responses in game G_1, with $A_{\widetilde{\mathsf{NIA}}}$ running its own GETPMG oracle on line 4. These queries to its own oracle are tracked as inputs in set $\mathcal{X}_x$. Now suppose that bad is set in G_1.

Then $\phi^* \notin \mathcal{X}_s$, $v((\phi^*, \pi^*)) = 0$ and $\widetilde{v}((\phi^*, \pi^*)) = 1$. In particular, $\phi^* \notin \mathcal{X}_s$ implies that $(\phi^*, \pi^*) \notin \mathcal{X}_x$. Now $x^* = (\phi^*, \pi^*)$ is precisely an input on which $v, \widetilde{v}$ differ, where x^* was never produced by the GETPMG oracle. This is the winning condition for adversary $A_{\widetilde{\mathsf{NIA}}}$ in game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{NIA},\widetilde{\mathsf{NIA}},\mathsf{P}_{\mathsf{pf}}}$, proving Eq. (14).

We next turn to adversary A_{NIA} which is in game $\mathbf{G}^{\mathrm{snd}}_{\mathsf{NIA}}$ and is depicted on the right side of Figure 19. A's view is that of game G_1; initialization and $\text{SIM}_{\mathsf{NIA}}$ return the same responses as in G_1. Now, if $\mathrm{G}_1(A)$ returns true, and since the boxed code is executed in G_1, the same conditions are satisfied as in $\mathbf{G}^{\mathrm{snd}}_{\mathsf{NIA}}$. Hence A returns a winning output which proves Eq. (15).

We conclude the proof by observing that $A_{\widetilde{\mathsf{NIA}}}$ and A_{NIA} maintain running times close to that of A. ∎

Games G_0, $\boxed{\mathrm{G}_1}$

INIT:
1 $(v, kg, s) \leftarrow\!\!\$\ \mathsf{TS}$; $(\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{TS}}(v)$
2 $(vk^*, sk^*) \leftarrow\!\!\$\ kg$; $\mathcal{X}_x \leftarrow \emptyset$; $\mathcal{X}_m \leftarrow \emptyset$
3 Return $(v, \widetilde{v}, vk^*, \{kg, s\})$

ESIGN(vk', m'):
4 $x \leftarrow\!\!\$\ e((vk', m'), 1)$; $(vk, m, \sigma) \leftarrow x$
5 $\mathcal{X}_x \leftarrow \mathcal{X}_x \cup \{x\}$; $\mathcal{X}_m \leftarrow \mathcal{X}_m \cup \{(vk, m)\}$
6 Return σ

SIGN(m):
7 $\sigma \leftarrow\!\!\$\ s(sk^*, m)$; $\mathcal{X}_m \leftarrow \mathcal{X}_m \cup \{(vk^*, m)\}$
8 Return σ

FIN(m^*, σ^*):
9 If $(vk^*, m^*) \in \mathcal{X}_m$ then return false
10 If ($v((vk^*, m^*, \sigma^*)) = 0$ and $\widetilde{v}((vk^*, m^*, \sigma^*)) = 1$) then $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{\text{return false}}$
11 Return $\widetilde{v}((vk^*, m^*, \sigma^*))$

Figure 20: Games $\mathrm{G}_0, \mathrm{G}_1$ for the proof of Theorem 7.1. G_1 contains the boxed code and G_0 does not.

Adversary $A_{\widetilde{\mathsf{TS}}}(v, \widetilde{v}, \alpha = \{kg, s\})$:
1 $(vk^*, sk^*) \leftarrow\!\!\$\ kg$; $\mathcal{X}_x \leftarrow \emptyset$; $\mathcal{X}_m \leftarrow \emptyset$
2 $(m^*, \sigma^*) \leftarrow A[\mathrm{ESIGN}_{\widetilde{\mathsf{TS}}}, \mathrm{SIGN}_{\widetilde{\mathsf{TS}}}](v, \widetilde{v}, vk^*, \{kg, s\})$
3 $x^* \leftarrow (vk^*, m^*, \sigma^*)$; Return x^*

Oracle $\mathrm{ESIGN}_{\widetilde{\mathsf{TS}}}(vk', m')$:
4 $x \leftarrow \mathrm{GETPMG}((vk', m'), 1)$; $\mathcal{X}_x \leftarrow \mathcal{X}_x \cup \{x\}$
5 $(vk, m, \sigma) \leftarrow x$; $\mathcal{X}_m \leftarrow \mathcal{X}_m \cup \{(vk, m)\}$
6 Return σ

Oracle $\mathrm{SIGN}_{\widetilde{\mathsf{TS}}}(m)$:
7 $\sigma \leftarrow\!\!\$\ s(sk^*, m)$; $\mathcal{X}_m \leftarrow \mathcal{X}_m \cup \{(vk^*, m)\}$
8 Return σ

Adversary $A_{\mathsf{TS}}(v, vk^*, \{kg, s\})$:
1 $(\widetilde{v}, e) \leftarrow\!\!\$\ \widetilde{\mathsf{TS}}(v)$; $\mathcal{X}_m \leftarrow \emptyset$
2 $(m^*, \sigma^*) \leftarrow A[\mathrm{ESIGN}_{\mathsf{TS}}, \mathrm{SIGN}_{\mathsf{TS}}](v, \widetilde{v}, vk^*, \{kg, s\})$
3 Return (m^*, σ^*)

Oracle $\mathrm{ESIGN}_{\mathsf{TS}}(vk', m')$:
4 $(vk, m, \sigma) \leftarrow\!\!\$\ e((vk', m'), 1)$; $\mathcal{X}_m \leftarrow \mathcal{X}_m \cup \{(vk, m)\}$
5 Return σ

Oracle $\mathrm{SIGN}_{\mathsf{TS}}(m)$:
6 $\sigma \leftarrow \mathrm{SIGN}(m)$; $\mathcal{X}_m \leftarrow \mathcal{X}_m \cup \{(vk^*, m)\}$
7 Return σ

Figure 21: Adversaries $A_{\widetilde{\mathsf{TS}}}$ (left) and A_{TS} (right) for the proof of Theorem 7.1.

C Proof of Theorem 7.1

Proof of Theorem 7.1: Consider game G_0 of Figure 20. The winning condition of the ffe game is contained on lines 9,11 of G_0 while the boxed code on line 10 is excluded. Otherwise G_0 is identical to the ffe game so that

$$\mathbf{Adv}^{\mathrm{ffe}}_{\mathsf{TS},\widetilde{\mathsf{TS}}}(A) = \Pr\left[\,\mathrm{G}_0(A)\,\right].$$

Now let us look at game G_1 and line 10. G_1 outputs false whenever the bad flag on line 10 is set. This happens when $\widetilde{v}((vk^*, m^*, \sigma^*))$ is true, but $v((vk^*, m^*, \sigma^*))$ is not. Games $\mathrm{G}_0, \mathrm{G}_1$ are

identical-until-bad and we have

$$\begin{aligned}\Pr[\,\mathrm{G}_0(A)\,] &= \Pr[\,\mathrm{G}_1(A)\,] + (\Pr[\,\mathrm{G}_0(A)\,] - \Pr[\,\mathrm{G}_1(A)\,]) \\ &\le \Pr[\,\mathrm{G}_1(A)\,] + \Pr[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,]\ .\end{aligned}$$

We next construct adversaries $A_{\widetilde{\mathsf{TS}}}, A_{\mathsf{TS}}$ such that the following two equations hold:

$$\Pr[\,\mathrm{G}_1(A) \text{ sets } \mathsf{bad}\,] \le \mathbf{Adv}^{\mathrm{exc}}_{\mathsf{TS},\widetilde{\mathsf{TS}},\mathsf{P}_{\mathsf{ff}}}(A_{\widetilde{\mathsf{TS}}}) \tag{16}$$

$$\Pr[\,\mathrm{G}_1(A)\,] \le \mathbf{Adv}^{\text{uf-cma}}_{\mathsf{TS}}(A_{\mathsf{TS}})\ , \tag{17}$$

which will complete the proof of Eq. (9) in the theorem statement.

We give adversary $A_{\widetilde{\mathsf{TS}}}$ in the left panel of Figure 21. $A_{\widetilde{\mathsf{TS}}}$ is in game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{TS},\widetilde{\mathsf{TS}},\mathsf{P}_{\mathsf{ff}}}$ and runs A, responding to oracle queries according to $\textsc{ESign}_{\widetilde{\mathsf{TS}}}$ and $\textsc{Sign}_{\widetilde{\mathsf{TS}}}$. In particular, $A_{\widetilde{\mathsf{TS}}}$ uses its own GetPmg oracle on line 4 to respond to ESign queries. With the key-generation and signing algorithms kg, s provided as auxiliary information, $A_{\widetilde{\mathsf{TS}}}$ selects its own challenge (vk^*, sk^*) and responds to A's Sign queries appropriately. The set $\mathcal{X}_m$ tracks verification keys and messages appearing in queries, matching $\mathcal{X}$ in the ffe game, while the set $\mathcal{X}_x$ tracks values x produced by the GetPmg oracle. Now suppose that bad is set in G_1.

Then $(vk^*, m^*) \notin \mathcal{X}_m$, $v((vk^*, m^*, \sigma^*)) = 0$ and $\widetilde{v}((vk^*, m^*, \sigma^*)) = 1$. Since $(vk^*, m^*) \notin \mathcal{X}_m$ it follows that $(vk^*, m^*, \sigma^*) \notin \mathcal{X}_x$. Therefore $x^* = (vk^*, m^*, \sigma^*)$ is exactly an input on which $v, \widetilde{v}$ differ, where x^* was never produced by the GetPmg oracle. This is $A_{\widetilde{\mathsf{TS}}}$'s winning condition in game $\mathbf{G}^{\mathrm{exc}}_{\mathsf{TS},\widetilde{\mathsf{TS}},\mathsf{P}_{\mathsf{ff}}}$, which proves Eq. (16).

Now consider adversary A_{TS} on the right panel of Figure 21. A_{TS} is in game $\mathbf{G}^{\text{uf-cma}}_{\mathsf{TS}}$ and runs A, responding to oracle queries via $\textsc{ESign}_{\mathsf{TS}}$ and $\textsc{Sign}_{\mathsf{TS}}$. On line 6, A_{TS} calls its own Sign oracle to respond to A's $\textsc{Sign}_{\mathsf{TS}}$ queries. Thus the view of A is game G_1. Now $\mathrm{G}_1(A)$ returns true only when $(vk^*, m^*) \notin \mathcal{X}_m$, $\widetilde{v}((vk^*, m^*, \sigma^*)) = 1$, and $v((vk^*, m^*, \sigma^*)) = 1$, where the latter follows from the fact that line 10 must not have returned false. Recall that the uf-cma winning condition checks whether $m^* \in \mathcal{Q}$, where $\mathcal{Q}$ tracks all Sign queries. If $(vk^*, m^*) \notin \mathcal{X}_m$ then necessarily $m^* \notin \mathcal{Q}$. The uf-cma winning condition finally checks whether $v((vk^*, m^*, \sigma^*)) = 1$ which, as above, is included in the winning condition of game G_1. Thus if A wins G_1 then A_{TS} wins its uf-cma game. This proves Eq. (17).

We conclude the proof by noting that $A_{\widetilde{\mathsf{TS}}}$ makes q_e GetPmg queries while A_{TS} makes q_s Sign queries, and both maintain running times close to that of A. ∎