True2F: Backdoor-resistant authentication tokens

Emma Dauterman
Stanford and Google

Henry Corrigan-Gibbs
Stanford

David Mazières
Stanford

Dan Boneh
Stanford

Dominic Rizzo
Google

Abstract. We present True2F, a system for second-factor authentication that provides the benefits of conventional authentication tokens in the face of phishing and software compromise, while also providing strong protection against token faults and backdoors. To do so, we develop new lightweight two-party protocols for generating cryptographic keys and ECDSA signatures, and we implement new privacy defenses to prevent cross-origin token-fingerprinting attacks. To facilitate real-world deployment, our system is backwards-compatible with today's U2F-enabled web services and runs on commodity hardware tokens after a firmware modification. A True2F-protected authentication takes just 57ms to complete on the token, compared with 23ms for unprotected U2F.

1 Introduction

Two-factor authentication has become a standard defense against weak passwords, keyloggers, and other types of malware. Universal Second Factor (U2F) hardware authentication tokens are an especially effective type of second-factor authenticator. Because these tokens cryptographically bind their authentication messages to a specific origin, they block phishing attacks to which other approaches, such as time-based one-time passwords (e.g., Google Authenticator), leave users exposed. These tokens run on simple, dedicated hardware, and so they present a much smaller attack surface than mobile-phone-based methods do. Since Google mandated in early 2017 that all of its employees use U2F authentication tokens, the company has not discovered a single instance of corporate credential theft [76].

But the introduction of new, and often opaque, hardware components into a system carries additional risks as well. If these components are poorly designed or even intentionally backdoored, they can *undermine* the security of an otherwise well-functioning system. A now-infamous bug in the design of Infineon-made cryptographic chips led to critical vulnerabilities in millions of TPMs, electronic passports, laptops, and hardware tokens [94]. Users of these flawed Infineon chips ended up *worse off* than those who just implemented their cryptographic routines using standard software libraries.

The consequences of faulty or backdoored hardware authentication tokens can be equally dire. If a token's randomness source is faulty—a common failure mode for low-cost devices [51, 67, 79, 126]—then anyone who phishes a user's credentials can authenticate as that user, even without access to the user's machine. An attacker who compromises a hardware token during manufacturing or via supply-chain tampering [37, 57, 98, 107] can exfiltrate the user's cryptographic credentials or authentication history (e.g., to an attacker-controlled web service). The latest batch of web standards [115, 119, 131] allow using hardware tokens for password-less "first-factor" authentication, which only exacerbates these dangers.

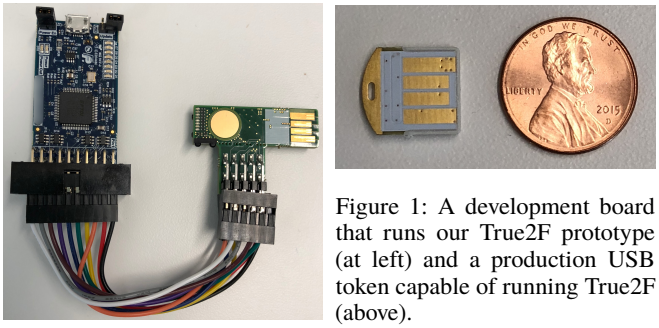

Figure 1: A development board that runs our True2F prototype (at left) and a production USB token capable of running True2F (above).

This raises the question: Can we enjoy the *benefits* of hardware-level security protections without incurring the *risks* of introducing additional (and often untrustworthy) hardware components into our systems? In this paper, we give a strong positive answer to this question for the interesting special case of hardware authentication tokens.

We present the design and implementation of True2F, a system for two-factor authentication that simultaneously provides the benefits of conventional second-factor authentication hardware tokens and very strong protection against the risks of token faults or backdoors. True2F requires only modest modifications to the token firmware and browser software. Critical for enabling incremental deployment is that True2F runs on *unmodified token hardware* and is backwards compatible with online services that already support U2F; using True2F requires *no server-side changes whatsoever*.

True2F uses the same authentication process as U2F: a relying party (i.e., web service) sends a challenge to the user's browser, the browser forwards the challenge to a hardware token via USB, and the token returns a signature on the challenge to the relying party via the browser. The only difference is that a True2F-compliant token and browser exchange a few extra messages before responding to the relying party. These messages allow the browser to enforce the token's correct behavior, preventing a malicious token from choosing weak or preloaded keys, or from covertly leaking keying material in messages to the relying party.

The True2F token provides the same level of protection against phishing and browser compromise as a standard U2F hardware authentication token does. After the user has registered the token with a relying party, even if the attacker takes control of the user's machine, the attacker still cannot authenticate to the relying party without interacting with the token. This holds even if the attacker can passively observe the browser's interactions with the token before taking control of the machine.

True2F additionally protects against token backdoors. If the browser executes the True2F protocol correctly, then even if the adversary controls a coalition of relying parties, the relying parties cannot detect whether they are interacting with an adversarial

U2F token or an ideal (honest) U2F token. In particular, the token cannot exfiltrate data to the outside world via U2F protocol messages.

With True2F, to compromise the cryptographic keys on the token, an attacker has to compromise *both* the user's machine and the hardware token itself. In this way, True2F provides "strongest-link" security, while standard U2F provides security that is only as good as the hardware token itself.

The design of True2F takes advantage of the fact that U2F hardware tokens perform only very simple computations: a U2F token generates keypairs for the ECDSA digital signature scheme, it signs server-provided challenges, and it keeps a counter to track how many authentications it has performed.

We develop lightweight two-party protocols that allow the token and browser to collaboratively execute all of these operations in such a way that (1) the browser learns nothing about the token's secrets and yet (2) the browser can enforce the token's strict compliance with the U2F specification. Our security definitions draw on the recent theoretical work on cryptographic firewalls [46, 87] and algorithm-substitution attacks [11].

In particular, we develop a new *collaborative key-generation protocol* that runs between the browser and token. The browser uses this protocol to ensure that the token's master keypair incorporates proper randomness. Our protocol is inspired by, but is roughly 3× faster than, a scheme of prior work [36].

We introduce *verifiable identity families*, a mechanism for deterministically deriving an exponential number of random-looking ECDSA keypairs from a single master keypair. We use this primitive in True2F to derive the per-website keypairs used for U2F authentication in a deterministic and browser-auditable way.

We also construct *firewalled ECDSA signatures*, which allow the browser and token to jointly produce ECDSA signatures on messages (1) without revealing any of the token's secrets to the browser and (2) while preventing the token from exfiltrating secrets via the bits of the signature. This prevents an adversarial token from tampering with the randomness used to generate the signatures and from encoding bits of secret information (e.g., the token's signing keys) in the randomness of the signature itself [127]. The innovation is that our protocol outputs unmodified ECDSA signatures, which is critical for backwards compatibility. In Section 9, we explain how our construction relates to *subliminal-free* [43, 44, 45, 108, 109] and *subversion-resistant* signature schemes [6].

Finally, we implement a *flash-friendly counting data structure* that we use to implement the U2F authentication counters. As we will describe, using finer-grained counters better protects the user from cross-site token-fingerprinting attacks. To make it practical to keep fine-grained counters, we propose a new log-structured counter design that respects the tight design and space constraints of our token's flash hardware.

We have implemented the full True2F protocol and run it on a hardware token used by Google employees (Figure 1). On this token, a True2F authentication completes in 57ms, compared with 23ms for a standard U2F authentication. Registering the True2F token with a new website takes 109ms, compared with 64ms for a standard U2F registration.

Secondary application: Hardware wallets. While our focus is on authentication tokens, the principles of True2F also apply to protecting cryptocurrency "hardware wallets" against backdoors [19, 107]. Hardware wallets perform essentially the same operations as U2F tokens: ECDSA key generation and signing. When using True2F in the context of hardware wallets, the user would store the token's master secret key in cold storage for backup. Otherwise, the application to hardware wallets is almost immediate, and we do not discuss it further.

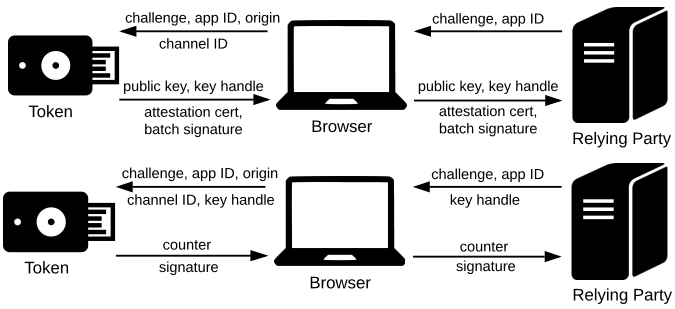

Figure 2: U2F registration (top) and authentication (bottom).

2 Background

The primary goal of the Universal Second Factor (U2F) [53, 78] standard is to defeat credential-phishing attacks [68] by requiring users of web services to authenticate not only with a human-memorable password ("the first factor") but also using a cryptographic secret stored on a hardware token ("a second factor"). Even if the user accidentally discloses their login credentials to an attacker, the attacker cannot access the user's account without also having physical access to the hardware token. All major web browsers support U2F, as do many popular web services such as Gmail, Facebook, Dropbox, and Github [115, 119, 129, 131].

Three entities make up a U2F deployment:

1. *a hardware token*, which is a small USB device, as in Figure 1, that stores cryptographic keys and computes digital signatures,
2. *a web browser* that serves as the user's interface to the token and the web, and
3. *a relying party* (i.e., service) to which the user wants to authenticate.

The U2F protocol specifies two actions: *registration* and *authentication*. Both actions begin when the relying party makes a U2F request to the browser via a JavaScript API. The browser annotates the request with connection metadata, forwards it to the U2F token via USB, and displays a prompt to the user, who physically touches the token to complete the request.

Registration. Registration associates a particular hardware token with a user's account at a relying party (e.g., `user@github.com`). During registration the relying party sends an application identifier (e.g., `github.com`) and a challenge to the token, as in Figure 2. The token then generates a "per-identity" ECDSA keypair $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}})$, and returns the public key $\mathsf{pk}_{\mathsf{id}}$ to the relying party via the browser. Since a single user may have multiple accounts with the same relying party, the identity id corresponds to an account-site pair, such as `user@github.com`.

The token also returns an opaque blob, called the "key handle," to the relying party. The relying party stores the public key $\mathsf{pk}_{\mathsf{id}}$ and key handle alongside the user's credentials. During

authentication, the relying party passes this key handle back to the token as part of its authentication request. The token can use the key-handle blob to reduce the amount of keying material it needs to store. For example, Yubico's U2F key [128] derives a per-identity secret key by applying a pseudorandom function [62], keyed with a global secret key, to the identity's key handle.

U2F also supports a simple form of hardware attestation. For privacy reasons, the Chrome browser essentially disables this feature and True2F follows Chrome's behavior [34].

Authentication. During authentication, the relying party sends to the browser the key handle produced at registration, the application identifier appID, and a challenge for the token to sign (Figure 2). The browser constructs a challenge with the relying party's challenge and connection metadata, and forwards the key handle and appID to the token. The token uses the key handle to recover the per-identity secret signing key $\mathsf{sk}_{\mathsf{id}}$. In addition, the token reads and increments a counter it stores locally. Finally, the token uses $\mathsf{sk}_{\mathsf{id}}$ to sign the browser's challenge along with the application ID and counter value.

The counter defends against physical cloning attacks. If an attacker clones a hardware token, then the original token and the clone will have the same counter value. If both tokens authenticate, then the relying party can notice that the counter has not increased (or possibly has decreased) and lock the account. U2F does not specify how the counter must be implemented—the only requirement is that the counter reported at each authentication attempt to a given relying party must be strictly increasing. Yubico's U2F key uses a single global counter [128].

3 Design overview

In this section, we sketch the high-level design ideas behind True2F and then describe our security goals for the system.

3.1 From a "Straw-man" to True2F

A simple way to prevent a backdoored hardware token from misbehaving would be to make its behavior deterministic: after purchasing the token, the user would load a single seed into the token. The token would then deterministically derive all of its randomness and cryptographic secrets from this seed using a pseudorandom generator. If the user stored the same cryptographic seed on the browser, the browser could mimic the behavior of the token, and could thus detect whether the token ever deviated from the U2F specification.

This "straw-man" solution has a number of critical drawbacks. First, to be able to audit the token, the browser must store the seed. If an attacker ever compromises the browser, the attacker can steal the seed and use it to learn all of the token's cryptographic secrets. Storing the seed on the browser would then completely defeat the purpose of having a hardware token in the first place.

We address this first issue using a new primitive that we call a *verifiable identity family* (Section 4.2). When the user initializes the token, she generates a master keypair $(\mathsf{msk}, \mathsf{mpk})$, consisting of a secret key and a public key. The user loads the master secret key onto the token and stores the master public key on the browser. When the browser initiates the U2F registration protocol for an identity id, the token then uses the master secret key msk to *deterministically* generate a unique (but random-looking) authentication keypair $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}})$. Using the master public key mpk, the browser can verify that $\mathsf{pk}_{\mathsf{id}}$ really is the unique public key that corresponds to identity id. In this way, the browser can verify that the token is generating its cryptographic keys according to our specification.

But even this solution is imperfect: if the user generates the master keypair $(\mathsf{msk}, \mathsf{mpk})$ on her computer, traces of the master secret key might be left in memory or swap files on her machine [32, 38, 58]. Malware that later compromises the browser could recover the master secret key msk and thereby learn all of the token's cryptographic secrets.

To address this second issue, we use a new *collaborative key-generation protocol* to generate the master keypair (Section 4.3). At the end of the protocol, the token holds the master secret key and the browser holds the master public key. The protocol further enforces that (a) the keypair is sampled from the correct distribution and (b) the browser learns nothing except the master public key. By using this protocol for key generation, the browser can ensure that the token uses a master key that incorporates sufficient randomness and the token ensures the browser never sees its cryptographic secrets.

Even these two improvements are still not quite sufficient to achieve a fully backdoor-resistant U2F token. The last remaining issue is the ECDSA signature that the token produces in response to each authentication request. Since ECDSA signatures are randomized, a faulty token could use bad randomness to generate the signatures. In addition, a malicious token could leak bits of information (e.g., about its master secret key) by hiding them in the bits of the signature [21, 28, 43, 44, 45, 108, 109].

To ensure that the token uses proper signing randomness and to eliminate covert channels in the signatures, we introduce *firewalled ECDSA signatures* (Section 4.4). To produce a firewalled signature, the token (holding a secret signing key sk and message m) and browser (holding a public verification key pk and message m) run an interactive protocol. At the end of the protocol, the browser either (a) holds an ECDSA signature on m that is indistinguishable from an honestly generated ECDSA signature using sk on m or (b) outputs "token failure." Crucially, our signing protocol is extremely efficient: it requires the token to perform only two exponentiations in the group, while ECDSA requires one.

Finally, we make one privacy improvement to the standard U2F token design. As described in Section 2, to prevent cloning attacks, the U2F token includes a counter value in each authentication message it sends to the relying party. To use a minimal amount of non-volatile storage on the token, today's tokens use a single global counter that increments after each authentication attempt. Using a global counter poses a privacy risk: a coalition of relying parties can use this counter value as a fingerprint to track a single user as she authenticates to different services around the Web.

Using per-identity counters would protect against fingerprinting, but could require a large amount of storage space. Because of hardware constraints we describe in Section 5, today's tokens use $2\times$ 2KB flash pages to store a single 23-bit counter. Using this strategy to implement 100 per-identity counters would require 400KB of non-volatile storage—consuming almost all of our token's 512KB flash bank.

To allow for the use of distinct per-identity counters while respecting the storage constraints of these weak tokens, we introduce a *flash-optimized counter* data structure (Section 5). Our design, which is inspired by log-structured file systems and

indexes [3, 4, 41, 42, 82, 102, 120, 132], can store 100 unique counters using only 6KB of flash, while still supporting over 2^{22} total authentications.

Putting these four pieces together yields the full True2F design.

3.2 Security goals

We now describe the security goals of the True2F design.

Protect against a malicious token. If the browser executes the protocol faithfully, the browser protects against arbitrary misbehavior by the token.

To make this notion more formal, we adopt a definition inspired by the work of Mironov and Stephens-Davidowitz on cryptographic reverse firewalls [87]. We say that the protocol protects against a malicious U2F token if an arbitrarily malicious relying party cannot distinguish the following two worlds:

W1 Server interacts with an "ideal" U2F token that executes the protocol faithfully.

W2 Server interacts with an arbitrarily malicious U2F token that (a) sits behind an honest browser and (b) never causes the honest browser to abort.

A protocol that is secure under this definition prevents a malicious token from exfiltrating secrets to a malicious relying party.

There are two nuances to this definition. First, this security definition does not capture a number of critical potential covert channels, including timing channels and other other out-of-protocol attacks. We discuss these attacks in Section 3.4.

Second, in our protocols, if the token sends a malformed message, the browser may output "Token Failure" and refuse to continue processing messages. We require that the relying party not be able to distinguish the two worlds described above *only for tokens that* ***never*** *cause an honest browser to abort* in this way. We restrict our attention to tokens that do not cause the browser to abort for two reasons:

- If the token fails in a detectable way, the user knows that something has gone wrong and can remediate by discarding the token and reinstalling the software on her machine.
- Furthermore, even if a malicious token triggers a failure event at a chosen time, it gains at most a negligible advantage in exfiltrating the user's cryptographic secrets to the outside world. In Appendix A, we explain why this is so.

Protect against a compromised browser. If the token executes the protocol faithfully, it should give the same level of protection against a malicious browser that U2F does today. In particular, we consider an adversary that can:

- passively observe the state of the browser at all times, and
- can take control of the browser at an adversary-chosen time T.

We require that such an adversary cannot authenticate to any sites that were registered with the token before time T, except by interacting with the token.

Allowing the attacker to observe the state of the browser at all times models the fact that it is notoriously difficult to erase secrets from modern computers [32, 38, 58]. Because of this, if an attacker compromises a browser at time T, it can likely recover bits of information about the browser's state from times $T' < T$.

Once the attacker takes control of the browser at time T, the attacker can register for new sites without interacting with the honest token (e.g., by forwarding U2F requests to an attacker-controlled token elsewhere). So, the most we can require is that the token must participate when authenticating to sites registered at times $T' < T$.

Protect against token fingerprinting. If the token and browser execute the protocol faithfully, then coalitions of malicious relying parties should not be able to "fingerprint" the token, with the aim of tracking the user across different websites or linking different pseudonymous accounts on the same web site.

We define tracking-resistance using a "real vs. ideal" security game between a challenger and an adversary. In the game, the challenger gives the attacker access to a *registration* oracle and an *authentication* oracle. These oracles implement the functionalities provided by the U2F token. The attacker may invoke the registration oracle at most I times (e.g., $I \approx 100$), which models an attacker that can coerce a user into registering its token under at most I distinct identities at attacker-controlled websites. The attacker may invoke the authentication oracle at most A times (e.g., $A \approx 2^{22}$), which represents an upper bound on the number of authentication attempts the attacker can observe.

In the real world, the challenger implements these oracles using a *single* U2F token. In the ideal world, the challenger implements these oracles using I *distinct* U2F tokens—one token per identity. We say that the token protects against I-identity A-authentication fingerprinting if no efficient adversary can distinguish these two worlds with probability non-negligibly better than random guessing.

The browser UI should allow the user to distinguish between U2F registration and authentication, otherwise a single malicious relying party could trick a user into registering their token more than I times, which could subvert True2F's fingerprinting guarantees. To prevent this attack, the browser could alert the user before the Ith registration attempt, or the browser could prevent more than I registrations with a single token.

When $I > 1$, this notion of tracking resistance is much stronger than today's U2F tokens provide, since the use of a global authentication counter in today's U2F tokens provides a convenient device fingerprint. We discuss this issue in detail in Section 5.

3.3 Functionality goals

To make any backdoor-resistant U2F design usable in practice, it should be backwards-compatible in the following ways.

No changes to the relying party (server). While a security-conscious user can upgrade their U2F token and browser to use our new backdoor-resistant design, a security-conscious user is *not* able to upgrade the software running on their bank or employer's web server. A security-conscious user should be able to protect herself against faulty or malicious U2F tokens without waiting for all of the relying parties on the Internet to upgrade to a new authentication protocol.

No changes to the U2F token hardware. While we might be able to achieve better performance by (for example) putting a faster processor in the token, by restricting ourselves to today's hardware, we make sure that the deployment cost stays low.

Using True2F with many browsers. To audit the token across multiple browsers, we can piggyback on existing mechanisms for syncing web bookmarks and preferences across a user's browser instances (e.g., Firefox Sync). The browser can store the token verification state—a total of a few kilobytes—alongside the data

that the browser already syncs. In this way, all of the user's browser instances can effectively audit a single U2F token.

The downside of this approach is that if an attacker manages to hijack any one of a user's browser instances (or the sync server), the attacker can corrupt the U2F verification state on all instances. This trade-off seems somehow inherent; users who want the strongest protection against browser and token compromise can always use a distinct U2F token on each machine.

3.4 Timing attacks and other covert channels

True2F eliminates all "in-protocol" covert channels that a malicious U2F token could use to exfiltrate information to a malicious relying party. This still leaves open the possibility of "out-of-protocol" covert channels, which we discuss here.

Timing. By modulating the amount of time it takes to respond to a relying party's authentication request, a malicious token could leak information to the relying party without changing the U2F protocol messages at all [69]. Essentially all exfiltration-prevention systems are vulnerable to some form of timing attack [29, 87], and ours is no different.

To partially defend against this class of attacks, the browser could add a random delay to U2F responses [69, 74]. The browser could also delay the response so that it always takes a fixed amount of time (e.g., five seconds) to complete. Another defense would be to send a registration or authentication request to the token only after the user presses a button on the token—rather than waiting for a button press after the request is sent. This modification would make it more difficult for a malicious relying party to accurately measure the amount of time the token takes to complete a request.

Failure. By selectively refusing to complete the authentication protocol, the token can leak roughly $\log_2(T+1)$ bits of information to an outside relying party over the course of T U2F interactions with the token. (See Appendix A.) To mitigate the damage of this attack, the user of a token should discard the token as faulty if it *ever* fails to complete an authentication request.

Physical attacks. If an adversary steals the token and can interact with it directly—without intermediation by the browser—then a malicious token could leak all of its secrets to the thief in response to a "magic input." We do not attempt to protect against such physical attacks.

Other. A backdoored U2F token could leak secrets outside of the flow of the U2F protocol using a built-in Bluetooth or GSM modem. Or, a backdoored U2F token might be able to masquerade as a keyboard or other input device, and coerce the browser or some other program on the user's machine into leaking secrets to the outside world [113]. Physical inspection of the token (e.g., to look for antennas) would detect the most egregious physical backdoors. Additional OS-level protections could defeat attacks via USB [5, 112, 130].

4 Cryptographic building blocks

We first recall a number of standard cryptographic building blocks and then introduce our new cryptographic protocols.

Notation. For an integer q, we use $\mathbb{Z}_q$ to denote the ring of integers modulo q and $\mathbb{Z}_q^*$ to denote its invertible elements. For a variable x, $x \leftarrow 3$ indicates assignment. For a finite set $\mathcal{S}$, $r \xleftarrow{R} \mathcal{S}$ denotes a uniform random draw from $\mathcal{S}$. We use "$\perp$" as a special symbol indicating failure. When $\mathbb{G} = \langle g \rangle$ is a finite group, we always write the group notation multiplicatively. So, an ECDSA public key has the form $X = g^x \in \mathbb{G}$.

An *efficient* algorithm is one that runs in probabilistic polynomial time. When A is a randomized algorithm, we use $A(x; r)$ to denote running A on input x with random coins r. A *negligible* function is one whose inverse grows faster than any fixed polynomial. All of the cryptographic routines we define take a security parameter as an implicit input and we require that they run in time polynomial in this parameter.

4.1 Standard primitives

Digital signatures. A digital signature scheme over a message space $\mathcal{M}$ consists of a triple of algorithms:

- $\mathsf{Sig.KeyGen}() \to (\mathsf{sk}, \mathsf{pk})$. Output a secret signing key sk and a public verification key pk.
- $\mathsf{Sig.Sign}(\mathsf{sk}, m) \to \sigma$. Output a signature σ on the message $m \in \mathcal{M}$ using the secret key sk.
- $\mathsf{Sig.Verify}(\mathsf{pk}, m, \sigma) \to \{0, 1\}$. Output "1" iff σ is a valid signature on message m under public key pk.

For all keypairs $(\mathsf{sk}, \mathsf{pk})$ output by $\mathsf{Sig.KeyGen}$, for all messages $m \in \mathcal{M}$, $\mathsf{Sig.Verify}(\mathsf{pk}, m, \mathsf{Sig.Sign}(\mathsf{sk}, m)) = 1$.

We use the standard security notion for signature schemes: *existential unforgeability under chosen message attack* [64]. Informally, no efficient adversary should be able to construct a valid signature on new a message of its choosing, even after seeing signatures on any number of other messages of its choosing.

The current U2F standard mandates the use of ECDSA signatures [53], summarized in Appendix B, so our protocols take the peculiarities of ECDSA into account. That said, our protocols are compatible with more modern signatures schemes, such as Schnorr-style [106] elliptic-curve signatures [14] and BLS [23].

Verifiable random function (VRF). We use VRFs [47, 60, 86] defined over input space $\mathcal{I}$ and output space $\mathbb{Z}_q^*$, for some prime integer q. In our application, the input space $\mathcal{I}$ is a set of "identities" (e.g., username-website pairs). A VRF is a triple of algorithms:

- $\mathsf{VRF.KeyGen}() \to (\mathsf{sk}_{\mathsf{VRF}}, \mathsf{pk}_{\mathsf{VRF}})$. Output a secret key and a public key.
- $\mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}) \to (y, \pi)$. Take as input the secret key $\mathsf{sk}_{\mathsf{VRF}}$, and an input $\mathsf{id} \in \mathcal{I}$, and output a value $y \in \mathbb{Z}_q^*$ along with a proof π.
- $\mathsf{VRF.Verify}(\mathsf{pk}_{\mathsf{VRF}}, \mathsf{id}, y, \pi) \to \{0, 1\}$. Take as input the public key $\mathsf{pk}_{\mathsf{VRF}}$, a purported input-output pair $(\mathsf{id}, y) \in \mathcal{I} \times \mathbb{Z}_q^*$, and a proof π. Return "1" iff π is a valid proof that (id, y) is an input-output pair.

To be useful, a VRF must satisfy the following standard notions, which we state informally. We refer the reader to prior work [91, 99] for formal definitions of these properties.

- **Completeness.** The $\mathsf{VRF.Verify}$ routine accepts as valid all proofs output by $\mathsf{VRF.Eval}$.
- **Soundness.** It is infeasible to find two input-output-proof triples (id, y, π) and (id, y', π') such that (a) $y \neq y'$ and (b) $\mathsf{VRF.Verify}$ accepts both triples.
- **Pseudorandomness.** Informally, even if the adversary can ask for VRF output-proof pairs $(y_{\mathsf{id}}, \pi_{\mathsf{id}})$ on ids of its choosing, the adversary cannot distinguish a VRF output y_{id^*} corresponding

to an unqueried input id^* from a random point in $\mathbb{Z}_q^*$ with probability non-negligibly better than $1/2$.

We use a variant of the VRF construction of Papadopoulos et al. [99], which is secure in the random-oracle model [12] under the Decision Diffie-Hellman assumption [22]. (We make only a trivial modification to their VRF construction to allow it to output elements of $\mathbb{Z}_q^*$, rather than group elements.)

The only special property that we use of this VRF construction is that its keypairs have the form $(x, g^x) \in \mathbb{Z}_q \times \mathbb{G}$, where $\mathbb{G} = \langle g \rangle$ is a group of prime order q, for x random in $\mathbb{Z}_q$. Since we otherwise make only black-box use of the VRF construction, we do not discuss it further.

4.2 New tool: Verifiable identity families (VIFs)

Today's U2F tokens authenticate to every website under a distinct per-identity public key. Our protocol needs to ensure that the token samples its per-identity public keys from the correct distribution (or, more precisely, a distribution indistinguishable from the correct one). To do so, we introduce and construct a new primitive, called a *verifiable identity family* (VIF).

The VIF key-generation routine outputs a master secret key and a master public key. The master public key is essentially a commitment to a function from identities $\mathsf{id} \in \mathcal{I}$ (i.e., username-website pairs) to public keys for a signature scheme Σ. Informally, the scheme provides the following functionality:

- Anyone holding the master secret key can produce the *unique* public key $\mathsf{pk}_{\mathsf{id}}$ corresponding to a particular identity $\mathsf{id} \in \mathcal{I}$, and can also produce the corresponding secret key $\mathsf{sk}_{\mathsf{id}}$ for the signature scheme Σ.
- The holder of the master secret key can prove to anyone holding the master public key that $\mathsf{pk}_{\mathsf{id}}$ is really the unique public key corresponding to the string id.

Before giving a more precise definition, we first explain how we use a VIF in our U2F token design. At the completion of the token initialization process, the browser holds a VIF master public key, and the U2F token holds a VIF master secret key. Whenever the user wants to register the token under a new identity (e.g., $\mathsf{id} = \texttt{user@example.com}$), the browser sends the identity id to the token, and the token returns a per-identity public key $\mathsf{pk}_{\mathsf{id}}$ to the browser, along with a proof that the public key was computed correctly. The browser verifies the proof to convince itself that $\mathsf{pk}_{\mathsf{id}}$ is the correct public key for $\mathsf{id} = \texttt{user@example.com}$ under the token's master public key. The soundness property of the VIF ensures that there is a *unique* public key that the browser will accept from the token for this site. In this way, we can prevent a malicious token from leaking bits of information to a malicious website via a non-random or otherwise malformed public key.

Furthermore, we require the VIF-generated public keys to be pseudorandom: a malicious browser should not be able to predict what the VIF-generated public key will be for an identity id without querying the token for $\mathsf{pk}_{\mathsf{id}}$. This prevents a malicious browser from registering the token at new websites without the token's participation.

Finally, we require the VIF to satisfy an unforgeability property: a malicious browser should not be able to forge signatures that verify under VIF-generated public keys. If the VIF satisfies unforgeability, a malicious browser cannot learn useful information about the token's secrets, even after it watches the token authenticate to many websites.

Hierarchical wallets [122], used in Bitcoin, are closely related to VIFs. The key difference is that hierarchical wallet schemes either (a) do not allow the holder of a public key $\mathsf{pk}_{\mathsf{id}}$ to verify that it is the unique key corresponding to a master public key mpk, or (b) do not satisfy Σ-pseudorandomness, defined below. Both properties are crucial for our application.

Syntax and definitions. A VIF is a triple of efficient algorithms, defined with respect to a signature scheme $\Sigma = (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$:

- $\mathsf{VIF.KeyGen}() \to (\mathsf{msk}, \mathsf{mpk})$. Output a master secret key msk and a master public key mpk.
- $\mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id}) \to (\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}, \pi)$. Given an identity string $\mathsf{id} \in \mathcal{I}$, output the keypair $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}})$ corresponding to that identity, as well as a proof π that $\mathsf{pk}_{\mathsf{id}}$ is well formed. The keypair $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}})$ must be a valid keypair for the signature scheme Σ.
- $\mathsf{VIF.Verify}(\mathsf{mpk}, \mathsf{id}, \mathsf{pk}_{\mathsf{id}}, \pi) \to \{0, 1\}$. Verify that $\mathsf{pk}_{\mathsf{id}}$ is the public key for identity $\mathsf{id} \in \mathcal{I}$ corresponding to master public key mpk using the proof π. Return "1" iff the proof is valid.

The last two algorithms are deterministic.

We require the following four security properties to hold. The first two properties are almost identical to the properties required from a VRF. The last two properties are new, so we define them formally in Appendix C.1.

- **Completeness.** For any $(\mathsf{msk}, \mathsf{mpk})$ output by $\mathsf{VIF.KeyGen}$ and any identity $\mathsf{id} \in \mathcal{I}$, if $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}, \pi_{\mathsf{id}}) \leftarrow \mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id})$ then $\mathsf{VIF.Verify}(\mathsf{mpk}, \mathsf{id}, \mathsf{pk}_{\mathsf{id}}, \pi_{\mathsf{id}}) = 1$.
- **Soundness.** For all efficient adversaries $\mathcal{A}$, if we sample $(\mathsf{msk}, \mathsf{mpk}) \leftarrow \mathsf{VIF.KeyGen}()$ and then run $\mathcal{A}(\mathsf{msk}, \mathsf{mpk})$, the probability that $\mathcal{A}$ outputs two identity-key-proof triples $(\mathsf{id}, \mathsf{pk}_{\mathsf{id}}, \pi)$ and $(\mathsf{id}, \mathsf{pk}'_{\mathsf{id}}, \pi')$ such that $\mathsf{pk}_{\mathsf{id}} \neq \mathsf{pk}'_{\mathsf{id}}$ and $\mathsf{VIF.Verify}(\mathsf{mpk}, \mathsf{id}, \mathsf{pk}_{\mathsf{id}}, \pi) = 1$ and $\mathsf{VIF.Verify}(\mathsf{mpk}, \mathsf{id}, \mathsf{pk}'_{\mathsf{id}}, \pi') = 1$, is negligible in the (implicit) security parameter.
- **Σ-Pseudorandomness.** Even if the adversary can see many Σ-type signature public keys $\{\mathsf{pk}_{\mathsf{id}_1}, \mathsf{pk}_{\mathsf{id}_2}, \dots\}$ for identities of its choosing, and even if the adversary can see Σ-type signatures on messages of its choosing using the corresponding secret keys $\{\mathsf{sk}_{\mathsf{id}_1}, \mathsf{sk}_{\mathsf{id}_2}, \dots\}$, the adversary still cannot distinguish the true public key $\mathsf{pk}_{\mathsf{id}^*}$ for some identity $\mathsf{id}^* \notin \{\mathsf{id}_1, \mathsf{id}_2, \dots\}$ from a fresh public key $\mathsf{pk}_{\mathsf{rand}}$ output by $\mathsf{Sig.KeyGen}()$ with probability non-negligibly more than $1/2$.

  This holds even if the adversary may ask for signatures under the secret key corresponding to the challenge public key (which is either $\mathsf{pk}_{\mathsf{id}^*}$ or $\mathsf{pk}_{\mathsf{rand}}$).
- **Σ-Unforgeability.** The keypairs output by $\mathsf{VIF.Eval}(\mathsf{msk}, \cdot)$ are safe to use as Σ-type signing keypairs. In particular, even if the adversary can see many Σ-type public keys output by $\mathsf{VIF.Eval}(\mathsf{msk}, \cdot)$ for identities of its choosing, and even if the adversary can see Σ-type signatures on messages $\{m_1, m_2, \dots\}$ of its choosing using the corresponding secret keys, the adversary still cannot produce a signature forgery, namely a triple $(\mathsf{id}^*, m^*, \sigma^*)$ such that (a) the adversary never asked for a signature on m^* under identity id^* and (b) σ^* is a valid signature on message m^* under the public key $\mathsf{pk}_{\mathsf{id}^*}$ that $\mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id}^*)$ outputs.

Our construction. Our construction of a VIF for the ECDSA

Our VIF construction for ECDSA. The ECDSA signature scheme uses a group $\mathbb{G}$ of prime order q. An ECDSA keypair is a pair $(y, g^y) \in \mathbb{Z}_q \times \mathbb{G}$, for $y \xleftarrow{R} \mathbb{Z}_q$. The construction makes use of a VRF $\mathcal{V} = (\mathsf{VRF.KeyGen}, \mathsf{VRF.Eval}, \mathsf{VRF.Verify})$ that maps $\mathcal{I}$ into $\mathbb{Z}_q^*$ and uses keypairs of the form $(x, g^x) \in \mathbb{Z}_q \times \mathbb{G}$. We instantiate the three VIF routines as follows:

- $\mathsf{VIF.KeyGen}() \to (\mathsf{msk}, \mathsf{mpk})$.
  - Choose a random $x \xleftarrow{R} \mathbb{Z}_q$.
  - Run $(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{pk}_{\mathsf{VRF}}) \leftarrow \mathsf{VRF.KeyGen}()$.
  - Output $\mathsf{msk} = (x, \mathsf{sk}_{\mathsf{VRF}})$, $\mathsf{mpk} = (g^x, \mathsf{pk}_{\mathsf{VRF}})$.
- $\mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id}) \to (\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}, \pi)$.
  - Parse the master secret key msk as a pair $(x, \mathsf{sk}_{\mathsf{VRF}})$.
  - Run $(y, \pi_{\mathsf{VRF}}) \leftarrow \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id})$.
  - Set $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}) \leftarrow (xy, g^{xy}) \in \mathbb{Z}_q \times \mathbb{G}$ and $\pi \leftarrow (y, \pi_{\mathsf{VRF}})$.
  - Output $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}, \pi)$.
- $\mathsf{VIF.Verify}(\mathsf{mpk}, \mathsf{id}, \mathsf{pk}_{\mathsf{id}}, \pi) \to \{0, 1\}$
  - Parse the master public key mpk as a pair $(X, \mathsf{pk}_{\mathsf{VRF}})$, where $X \in \mathbb{G}$. Parse the public key $\mathsf{pk}_{\mathsf{id}}$ as a group element $Y \in \mathbb{G}$. Parse the proof π as a pair (y, π_{VRF}).
  - Output "1" iff (a) $Y = X^y \in \mathbb{G}$ and (b) $\mathsf{VRF.Verify}(\mathsf{pk}_{\mathsf{VRF}}, \mathsf{id}, y, \pi_{\mathsf{VRF}}) = 1$.

Figure 3: Our verifiable identity family construction for ECDSA.

signature scheme appears as Figure 3. To sketch the idea behind the construction: the master public key consists of a standard ECDSA public key $X = g^x$ and a VRF public key. To generate the public key for an identity $\mathsf{id} \in \mathcal{I}$, the VIF evaluator applies a VRF to id to get a scalar $y \in \mathbb{Z}_q$. The public key for identity id is then $X^y \in \mathbb{G}$.

The completeness and soundness of this VIF construction follow immediately from the completeness and soundness of the underlying VRF. We prove the following theorem in Appendix C.2, as Theorems 4 and 5:

Theorem 1 (Informal)**.** *The VIF construction of Section 4.2 satisfies Σ-pseudorandomness and Σ-unforgeability in the random-oracle model, when Σ is the Idealized ECDSA signature scheme of Appendix B, instantiated with a secure VRF.*

Our VIF construction is also secure when used with the Schnorr digital signature scheme [106]. The proof of security follows a very similar structure to that in Appendix C.2.

Proving that our VIF construction satisfies Σ-unforgeability when Σ is the ECDSA signatures scheme is not entirely trivial. The reason is that when using our VIF construction to generate secret keys for a sequence of identities $\mathsf{id}_1, \mathsf{id}_2, \ldots, \mathsf{id}_n$, the corresponding secret keys are *related*. That is, the secret ECDSA signing keys are of the form: $(\mathsf{sk}_{\mathsf{id}_1}, \mathsf{sk}_{\mathsf{id}_2}, \ldots, \mathsf{sk}_{\mathsf{id}_n}) = (xy_1, xy_2, \ldots, xy_n)$, such that the attacker knows $y_1, y_2, \ldots, y_n \in \mathbb{Z}_q$. To prove that our VIF scheme satisfies Σ-unforgeability, we must prove that using ECDSA to sign messages with related keys in this way does not, for example, allow the attacker to recover the master secret key x. Morita et al. [89] show attacks on Schnorr and DSA-style signatures schemes when attacker may *choose* the relations $y_1, \ldots, y_n \in \mathbb{Z}_q$. In our case, the y-values are sampled using a VRF, so their attacks do not apply.

4.3 New tool: Fast collaborative key generation

Corrigan-Gibbs et al. present a protocol that allows a network router and certificate authority (CA) to collaboratively generate an ECDSA keypair [36]. By directly applying their result, we get a protocol in which the browser and token can jointly generate an ECDSA keypair in such a way that (1) the browser learns nothing about the secret key and (2) the token cannot bias the public key that the protocol outputs.

We present a protocol that achieves the same security properties as theirs, but that requires the token to compute only a single exponentiation in the group $\mathbb{G}$, while theirs requires at least three. Since our protocol runs on a computationally limited U2F token, this factor of three yields a non-trivial speedup.

The master keypair for our VIF construction of Section 4.2 consists of two ECDSA-style keypairs, so the browser and token can use this protocol to jointly generate the VIF master keypair.

The protocol is parameterized by a group $\mathbb{G}$ of prime order q. At the end of the protocol, the browser outputs an ECDSA public key $X \in \mathbb{G}$ and the token either (a) outputs an ECDSA secret key $x \in \mathbb{Z}_q$ such that $X = g^x$, or (b) outputs the failure symbol "$\perp$." We use the standard notion of *statistical closeness* of probability distributions [61]. If two distributions are statistically close, no efficient adversary can distinguish them.

The protocol maintains the following properties:

- **Completeness.** If both parties are honest, the token never outputs the failure symbol "$\perp$."
- **Bias-free.** At the conclusion of a protocol run, the following two properties hold:
  - If the browser is honest, it holds a public key drawn from a distribution that is statistically close to uniform over $\mathbb{G}$.
  - If the token is honest, it holds a private key drawn from a distribution that is statistically close to uniform over $\mathbb{Z}_q$, provided that the browser *never* causes the token to output "$\perp$."
- **Zero knowledge (Malicious browser learns nothing).** If the token is honest, the browser learns nothing from participating in the protocol except the output public key. Formally, there exists an efficient simulator that can simulate a malicious browser's view of the protocol given only the public key as input, provided that the browser never causes the token to output "$\perp$."

  With more work, we can give a relaxed definition for browsers that cause the token to output $\perp$ with noticeable probability. The extension is straightforward, so we use the simpler definition.

Protocol. The protocol can use any non-interactive statistically hiding commitment scheme [40, 90], but for simplicity, we describe it using a hash function $H : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_q$ that we model as a random oracle [12].

1. The browser commits to a random value $v \in \mathbb{Z}_q$. That is, the browser samples $v \xleftarrow{R} \mathbb{Z}_q$ and $r \xleftarrow{R} \mathbb{Z}_q$ and sends $c \leftarrow H(v, r)$ to the token.
2. The token samples $v' \xleftarrow{R} \mathbb{Z}_q$ and sends $V' \leftarrow g^{v'} \in \mathbb{G}$ to the browser.
3. The browser opens its commitment to v by sending (v, r) to the token. The browser outputs $X \leftarrow V' \cdot g^v \in \mathbb{G}$ as the public key.

4. The token checks that $H(v, r) = c$. If so, the token accepts and outputs $x \leftarrow v + v' \in \mathbb{Z}_q$. Otherwise, the token outputs "$\perp$."

Completeness follows immediately. In Appendix D, we prove:

Theorem 2 (Informal)**.** *The key-generation protocol above is bias-resistant and zero knowledge, when we model H as a random oracle.*

4.4 New tool: Firewalled ECDSA signatures

During authentication, the relying party sends a challenge string to the U2F token, and the token signs it using ECDSA. We want to ensure that the token cannot exfiltrate any secrets by embedding them in the bits of the signature. If the U2F standard used a *unique* signature scheme, such as RSA Full Domain Hash or BLS [23], this would be a non-issue. Since ECDSA is not unique, the token could use its signatures to leak its secrets.

To eliminate exfiltration attacks, we introduce *firewalled signatures*, inspired by cryptographic reverse firewalls [87]. A firewalled signature scheme is a standard digital signature scheme Σ, along with an interactive signing protocol that takes place between a signer $\mathcal{S}$ and a firewall $\mathcal{F}$. In a run of the signing protocol, the signer takes as input a secret signing key sk for Σ and a message m. The firewall takes as input the public key pk corresponding to sk and the same message m.

At the conclusion of the protocol, the firewall holds a signature σ on m that validates under the public key pk, and the firewall learns nothing else. Furthermore, σ is indistinguishable from an "ideal" Σ-type signature on the message m using the secret key sk. In this way, the signer cannot exfiltrate any bits of information in σ itself.

Definition and syntax. A firewalled signature scheme consists of a standard digital signature scheme $\Sigma = (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$ over a message space $\mathcal{M}$, along with an interactive protocol between a signer $\mathcal{S}$ and a firewall $\mathcal{F}$. For bitstrings $x, y \in \{0,1\}^*$, let $[\mathcal{S}(x) \leftrightarrow \mathcal{F}(y)]$ denote the string that $\mathcal{F}(y)$ outputs when interacting with $\mathcal{S}(x)$. Let $\mathsf{View}_{\mathcal{F}}[\mathcal{S}(x) \leftrightarrow \mathcal{F}(y)]$ denote the transcript of messages that $\mathcal{F}$ observes in this interaction, along with $\mathcal{F}$'s input.

Then the signing protocol $(\mathcal{S}, \mathcal{F})$ must satisfy:

- **Correctness.** For all $(\mathsf{sk}, \mathsf{pk})$ output by $\mathsf{Sig.KeyGen}$, for all messages $m \in \mathcal{M}$, if $\sigma \leftarrow [\mathcal{S}(\mathsf{sk}, m) \leftrightarrow \mathcal{F}(\mathsf{pk}, m)]$, then $\mathsf{Sig.Verify}(\mathsf{pk}, \sigma, m) = 1$.
- **Exfiltration resistance.** Informally, as long as a malicious signer $\mathcal{S}^*$ never causes the honest firewall $\mathcal{F}$ to reject, then no efficient algorithm can tell whether a signature σ was generated using the standard signing algorithm or via the interactive protocol between $\mathcal{S}^*$ and $\mathcal{F}$.

  Let $\mathcal{S}^*$ be an efficient algorithm such that for all messages $m \in \mathcal{M}$, for all choices of the random coins of $\mathcal{S}^*$ and $\mathcal{F}$, we have that $[\mathcal{S}^*(\mathsf{sk}, m) \leftrightarrow \mathcal{F}(\mathsf{pk}, m)] \neq \perp$. Then the following distributions are computationally indistinguishable

$$\mathcal{D}_{\text{real}} = \left\{\sigma \leftarrow [\mathcal{S}^*(\mathsf{sk}, m) \leftrightarrow \mathcal{F}(\mathsf{pk}, m)]\right\}$$
$$\mathcal{D}_{\text{ideal}} = \left\{\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}, m)\right\},$$

  where we sample $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Sig.KeyGen}()$.
- **Zero knowledge.** Informally, the only thing that a malicious firewall can learn from participating in the protocol is a valid signature on the message m.

Signing protocol for firewalled ECDSA. For a keypair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{ECDSA.KeyGen}()$, and a message $m \in \mathcal{M}$ in the ECDSA message space $\mathcal{M}$, the protocol takes place between a signer $\mathcal{S}(\mathsf{sk}, m)$ and a firewall $\mathcal{F}(\mathsf{pk}, m)$:

1. The parties $\mathcal{S}$ and $\mathcal{F}$ engage in the key-generation protocol of Section 4.3. The signer plays the role of the token and the firewall plays the role of the browser. If the protocol aborts, $\mathcal{F}$ outputs $\perp$. Otherwise, at the end of the protocol, the signer $\mathcal{S}$ holds a value $r \in \mathbb{Z}_q$ and the firewall $\mathcal{F}$ holds value $R = g^r \in \mathbb{G}$.
2. The signer $\mathcal{S}$ executes $\sigma \leftarrow \mathsf{ECDSA.Sign}(\mathsf{sk}, m; r)$, using $r \in \mathbb{Z}_q$ as the signing nonce, and sends the signature σ to $\mathcal{F}$.
3. The firewall uses Equation (1) of Appendix B to compute the value $R_{\text{abs}} \in \{g^{r'}, g^{-r'}\}$, where r' is the signing nonce used to generate σ. The firewall ensures that $\mathsf{ECDSA.Verify}(\mathsf{pk}, m, \sigma) = 1$ and $R \in \{R_{\text{abs}}, 1/(R_{\text{abs}})\}$, and outputs $\perp$ if either check fails.
4. As described in Appendix B, given a valid ECDSA signature σ on a message m under public key pk, anyone can produce a second valid signature $\bar{\sigma}$. The firewall $\mathcal{F}$ outputs a random signature from the set $\{\sigma, \bar{\sigma}\}$.

Figure 4: Our protocol for generating firewalled ECDSA signatures.

Let $\mathcal{F}^*$ be an efficient firewall that never causes the honest signer $\mathcal{S}$ to output "$\perp$." (As in Section 4.3, handling the more general case of arbitrary efficient $\mathcal{F}^*$ is straightforward.) Then, there exists an efficient algorithm Sim such that for all such $\mathcal{F}^*$ and and all messages $m \in \mathcal{M}$, the following distributions are computationally indistinguishable:

$$\mathcal{D}_{\text{real}} = \left\{\mathsf{View}_{\mathcal{F}^*}[\mathcal{S}(\mathsf{sk}, m) \leftrightarrow \mathcal{F}^*(\mathsf{pk}, m)]\right\}$$
$$\mathcal{D}_{\text{ideal}} = \left\{\mathsf{Sim}\big(\mathsf{pk},\ m,\ \mathsf{Sig.Sign}(\mathsf{sk}, m)\big)\right\},$$

where we sample $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Sig.KeyGen}()$.

Construction. It is possible to construct firewalled signatures for any signature scheme using pseudo-random functions and zero-knowledge proofs. We develop a much more efficient special-purpose signing protocol that enables us to firewall plain ECDSA signatures without resorting to general zero-knowledge techniques.

The ECDSA signature scheme is a tuple of algorithms $(\mathsf{ECDSA.KeyGen}, \mathsf{ECDSA.Sign}, \mathsf{ECDSA.Verify})$, whose definition we recall in in Appendix B. Recall that ECDSA signatures use a fixed group $\mathbb{G} = \langle g \rangle$ of prime order q.

There are two key ideas behind our construction:

- First, the only randomness in an ECDSA signature comes from the signer's random choice of a "signing nonce" $r \xleftarrow{R} \mathbb{Z}_q$. (In fact, r is sampled from $\mathbb{Z}_q^* = \mathbb{Z}_q \setminus \{0\}$, but this distinction is meaningless in practice, since the two distributions are statistically indistinguishable.)
- Second, given an ECDSA public key pk, a message m, and a valid ECDSA signature σ on m for pk, anyone can recover a value R_{abs}, such that, if $r \in \mathbb{Z}_q$ is the signing nonce, then either $R_{\text{abs}} = g^r$ or $R_{\text{abs}} = g^{-r}$. (See Equation (1) in Appendix B.)

In our signing protocol, the signer and firewall can use the collaborative key-generation protocol of Section 4.3 to generate this (r, g^r) pair in a way that ensures that r is distributed uniformly at random over $\mathbb{Z}_q$. (The firewall actually only recovers $R_{\text{abs}} = g^{\pm r}$, but we can patch this with one extra step.) The signer

then uses the random value r to produce the ECDSA signature. Finally, the firewall checks that the signature is valid and that the signer used the correct nonce r to generate it, and then the firewall outputs a rerandomized version of the signature. Figure 4 describes the full protocol.

We prove the following theorem in Appendix E:

Theorem 3 (Informal)**.** *If the key-generation protocol of Section 4.3 is bias free and zero knowledge, then the protocol of Figure 4 is exfiltration resistant and zero knowledge.*

5 Storing many counters in little space

As described in Section 2, the token must send a counter value to the relying party that increases with each authentication.

The simplest implementation, and the one used on Yubico's U2F token (and originally on our token as well), keeps a global counter that increments with each authentication attempt. However, using a single global counter—rather than a counter per identity—entails both security and privacy risks:

1. *Failure to always detect clones.* Say that an attacker clones a token and logs into site S_1 then S_2. Then say that the token owner (with the true token) logs into site S_2 then S_1. If the global counter had value c when the token was cloned, S_1 and S_2 will each see a counter value of $c + 1$ followed by $c + 2$, and neither site will detect that the token was cloned.
2. *Token fingerprinting.* A single global counter value serves as a token fingerprint that can allow colluding sites to track a user across sites. Consider a token that authenticates at colluding sites S_1, S_2, and S_1 again. If these sites see the counter values $(c, c + 1, c + 2)$, for some value c, during these three authentication attempts, the sites can make an educated guess that the same user is authenticating at these two sites.

5.1 Goals and hardware constraints

We need a data structure with an operation $\mathsf{Count.Init}() \to \mathsf{st}$ that initializes a fresh counter state, and an operation $\mathsf{Count.Inc}(\mathsf{st}, \mathsf{id}) \to (\mathsf{st}_{\mathsf{new}}, c_{\mathsf{id}})$, that increments the counter value associated with string $\mathsf{id} \in \{0,1\}^*$ and returns the new state $\mathsf{st}_{\mathsf{new}}$ and counter value c_{id}.

The functionality goals for our counter scheme are parameterized by a maximum number of identities I used and a maximum number of increments A (corresponding to A authentications). Informally, we would like the following properties:

- **No worse than a global counter.** Counter values always increase. Also, after performing t increments from a fresh state with any sequence of ids, the counter value returned (for any id) is never larger than t.
- **As good as I independent counters.** As long as the user authenticates to at most I sites, the structure behaves as if each identity id is assigned an independent counter.

Flash hardware constraints. Any counter scheme we design must conform to the constraints of our token's flash hardware. Typical low-cost U2F tokens have similar requirements.

Our token uses two 256KB banks of NOR flash, divided into 2KB pages of 32-bit words. The flash controller supports random-access reads, word-aligned writes, and page-level erases. In flash, an erase sets all of the bits in the page to `1`, and subsequent writes can only shift bits to `0` or leave them unchanged. An interrupted write or erase operation leaves the modified bits in an indeterminate state.

Because of the physics of the flash, the hardware allows

- only **erasing an entire page** at a time—it is not possible to erase only a single word,
- at most **eight writes to each 32-bit word** between erases, and
- at most **50,000 program/erase cycles per page** over the lifetime of the token, and
- any number of reads between erases.

Furthermore, the counter implementation must be robust to interrupts—if the user yanks the U2F token out of their machine during authentication, the counter should still end up in a coherent state and the counter value presented to any site should be greater than the previous value. This prevents a user from being locked out of their account.

5.2 A "straw-man" counter scheme

These implementation constraints make it surprisingly expensive, in terms of flash space, to store even a small counter.

Using a single flash page, we can implement a counter that counts up to 4,096. To do so, we encode the counter value in *unary*: the counter begins in the erased state (all `1`s). To increment the counter, we zero out ("knock down") the first four non-zero bits in the page. In this way, we respect the constraint of at most eight writes per 32-bit word and we never have to erase the page until the counter reaches its maximum value. There are 512 words per page, so we can count up to a maximum value of $512 \cdot (32/4) = 2^{12}$.

Putting two such counters together—one for the low-order bits and one for the high-order bits—yields a two-page counter that counts up to 2^{24}, or roughly 16 million. With a little more work, we can make this design robust to power failures and other interrupts, allowing us to count up to 2^{23}. Popular U2F tokens today use a very similar two-page counter design.

As discussed above, one way to provide better cloning protection and unlinkability is to use per-identity counters. If we wanted to have 100 per-identity counters on the token, this would require ***two hundred*** flash pages, or over 400KB of flash storage—on a token with only 512KB of storage total. In contrast, our design stores 100 counters that can support a total of over 2^{22} increment operations using only ***three*** flash pages.

5.3 New tool: Flash-friendly counters

In True2F, we implement the authentication counters using a logging data structure [102]. The structure conceptually consists of a fixed-length table of identity count pairs $\langle \mathsf{id}, c_{\mathsf{id}} \rangle$, along with a variable-length list of identities (the "log"). To increment the counter associated with identity id, we append id to the log. When the log grows large, we run a "garbage collection" operation: we use the log to update the table of identity-count pairs, and then erase the log.

Our implementation uses three flash pages—a log page and two data pages—and it can store up to $I = 100$ distinct counters and supports $A = 6.4$ million authentications. We give a graphical depiction of the structure in Figure 5.

Each data page stores a table of I identity-count pairs, along with an "overflow" counter c_{overflow} and a page serial number. The data page with the larger serial number is the *active* page,

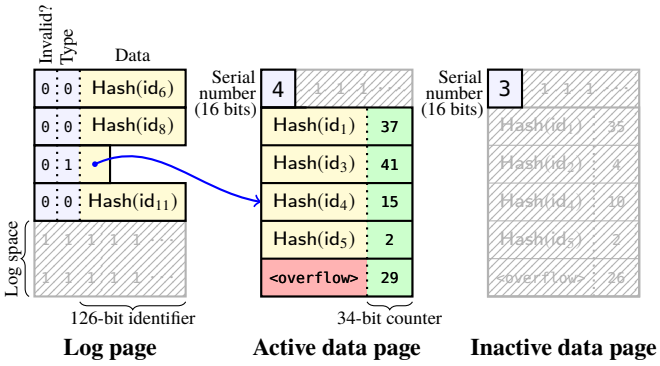

Figure 5: Counter data structure. The log stores hashes of identifiers (type 0) and pointers to entries in the active data page (type 1).

and the other data page is the *inactive* page. The log page stores the log. As a space-saving optimization, we actually only store hashes of identities, using a hash output large enough to avoid collisions in sets of at most I identities.

The counter value associated with identity id is the sum of (a) the number of times id appears in the log and (b) the counter value c_{id} associated with id in the active data page (if one exists) or the value c_{overflow} (if none exists).

To increment the counter associated with identity id, we can simply append the the string "id" to the end of the log page. To save space, when incrementing the counter associated with an identity id that already appears in the active data page, we write a pointer into the log that references the location of id in the active page. In the common case, when the user repeatedly authenticates under the same few identities, this optimization allows us to fit 8× more identities in the log.

The only complexity arises when the log page is nearly full. When this happens, we "garbage collect" the log: We pick I identities, first from the most recently used identities in the log page, and any remaining from the identities associated with the largest counter values in the active data page. We write these identity-count pairs into the inactive data page along with the new value of the overflow counter c_{overflow}. We compute this value as the maximum of the old overflow value and the counts associated with any ids that we "evicted" from the active data page during garbage collection. We then increment the serial number on the inactive data page, which makes it the new active page, and we erase the log page.

We order these operations carefully to ensure that even if the token loses power during these operations, it can later recover the correct counter state. To detect failures during log updates, each log entry is encoded with an "invalid" bit. When the token writes the last word of the log entry, it clears this bit to mark the entry as valid. To detect failures during garbage collection, the token writes a special symbol to the log before garbage collection with the serial number of the active page.

To compute the maximum number of increments: we erase each page at most once during garbage collection. In the worst case, where every authentication increments a unique counter value, the maximum number of increments is 50,000 (the maximum number of erases) times 128 (the number of hashes that can fit in the log), which yields 6.4 million increments total. In the best case, where we only increment at most $I = 100$ counters, the maximum number of increments is 128 (the number of hashes we add to the log before garbage collecting the first time) plus 49,999 (the maximum number of erases minus one) times 1,024 (the number of pointers that can fit in the log), for a total of 51 million increments. To increase this maximum value, a manufacturer could use multiple log pages.

6 The complete True2F protocols

We show how to assemble the cryptosystems of Section 4 with the counter design of Section 5 into the full True2F system (Figure 6).

I. Token initialization. True2F adds an initialization step to the U2F protocol. In this step, the True2F token and browser collaboratively generate a master keypair $(\mathsf{msk}, \mathsf{mpk})$ for our verifiable identity family construction from Section 4.2. Since a VIF keypair for our construction consists of two ECDSA-style keypairs (one for signing and one for the VRF), the token and browser can generate both keypairs using our key-generation protocol of Section 4.3. When using our protocol to generate the master VIF keypair:

- an honest browser is assured it is using a master public key sampled from the correct distribution and
- an honest token is assured that the browser learns nothing about its master secret key (other than what can be inferred from the master public key).

Alternative: Load keys from an external source. A user could generate the master VIF keypair $(\mathsf{msk}, \mathsf{mpk})$ on a single machine—separate from both the token and browser—and could load msk onto the True2F token and mpk onto the browser. If the user puts msk in offline storage, she can use it to recover all of her on-token secrets if she loses the physical token.

The browser and token also both initialize a counter data structure (Section 5) to store the U2F authentication counters. If both parties are honest, then the browser's counter state will replicate the state on the token.

II. Registration with website. During registration, the relying party sends the browser an application identity (e.g., service name) and a random challenge, as shown in Figure 2. The token must return (1) a "key handle" that the relying party will pass back to the token during authentication, (2) a public key, (3) an attestation certificate containing the attestation public key, and (4) a signature over the registration message with the attestation secret key corresponding to the attestation public key. The last two fields are used for U2F's hardware attestation feature. For privacy reasons, the Chrome browser by default generates a fresh random self-signed attestation certificate on every registration and uses this to sign the registration message [34]. True2F follows Chrome's design.

In True2F, the browser chooses a random key handle $\mathsf{id} \xleftarrow{R} \{0,1\}^{256}$ during registration. Using a random key handle ensures that if the user registers twice at the same website, the user receives two independent identities. The browser then sends id to the token and the token uses the VIF and its master secret key to derive the unique public key $\mathsf{pk}_{\mathsf{id}}$ corresponding to this identity. The token returns this key along with a proof of correctness π to the browser. The browser then checks the proof using the VIF master public key.

III. Authentication with website. During authentication, the relying party sends the browser (1) an application identifier,

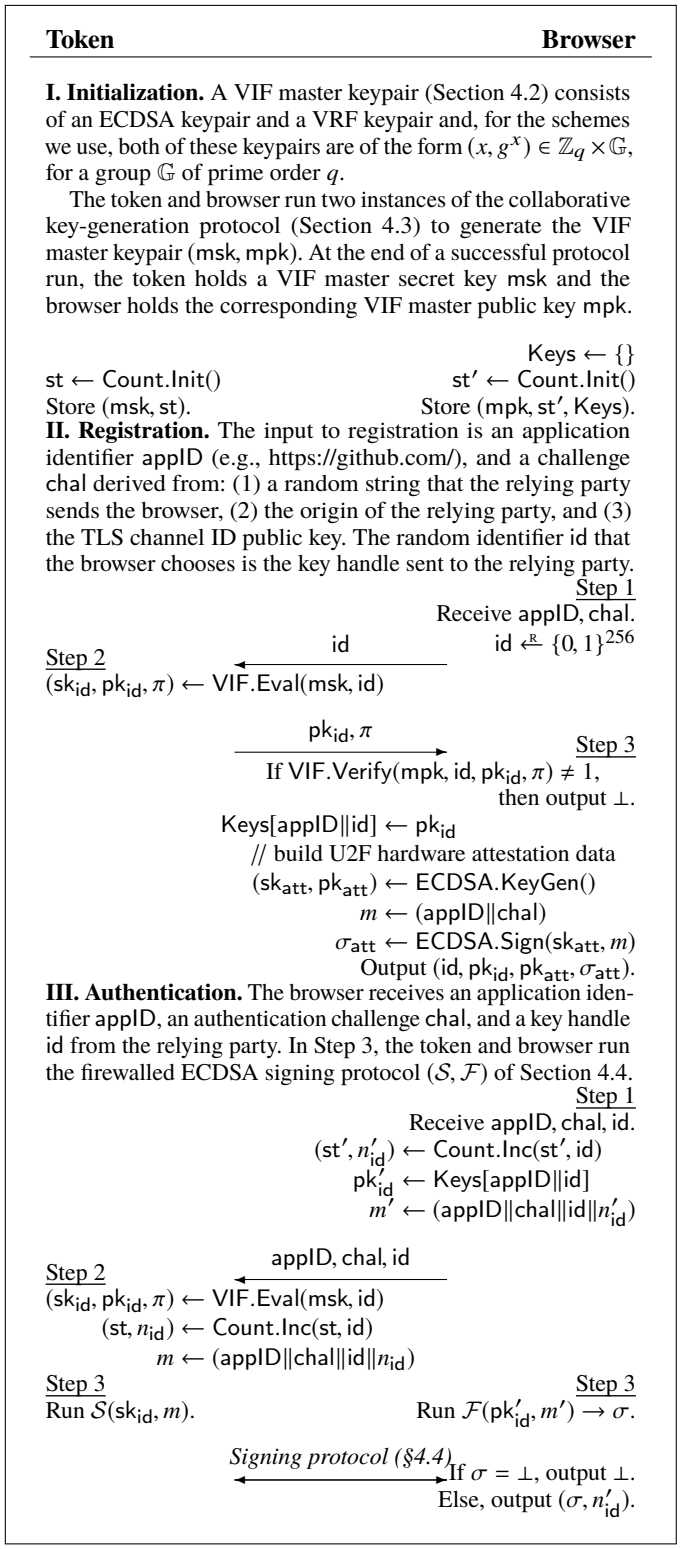

Figure 6: An overview of the True2F protocols. For clarity, the figure does not describe the optimizations of Section 7.2, and omits the U2F control fields ("control byte" and "user-presence byte").

(2) a challenge, and (3) a key handle, as shown in Figure 2. The token must return a counter value and a signature over the authentication request and counter value.

In True2F, the browser sends the application identifier, challenge, and key handle to the token. The token then uses the VIF to derive the per-identity signing key $\mathsf{sk}_{\mathsf{id}}$ for this identity and the browser can recover the per-identity public key $\mathsf{pk}_{\mathsf{id}}$ it received from the token during registration. The browser and token can both use their respective counter state to derive the counter value for identity (i.e., key handle) id. Using the application identifier, challenge, and counter value, the browser and token can both derive the message m to be signed.

The browser and token then engage in the firewalled ECDSA signing protocol of Section 4.4 to jointly produce a signature σ on the challenge message m. If the signing protocol succeeds, the browser returns the signature σ and counter value n'_{id} to the relying party.

One detail is that the message that the token signs includes a "user-presence byte" that indicates whether the user physically touched the token during authentication. The U2F v1.2 specification [53] has a mode in which the user need not touch the token during authentication. To prevent data leakage via the user-presence byte, the browser must ensure that it is set to `0x01` if the relying party required user presence and `0x00` otherwise.

Security analysis. We argue informally that our protocol provides the three security properties defined in Section 3.2. We discussed timing and selective failure attacks in Section 3.4.

We prove each claimed property from Section 3.2 using a hybrid argument. We begin with the real interaction between adversary and challenger, as defined in that section. We then make a number of incremental modifications to this interaction, arguing at each step that the adversary cannot distinguish the modified interaction from the real one. Finally, we reach an "ideal" interaction, in which the adversary cannot break security, which completes the proof.

Property 1: True2F protects against a malicious token.
Proof idea: The proof proceeds in four steps. We first replace the master public key mpk obtained by the browser with one generated by VIF.KeyGen(). The bias-freeness of the key-generation protocol permits this change.

Then, we replace the per-identity public keys $\mathsf{pk}_{\mathsf{id}}$ sent by the token during registration with ones generated by VIF.Eval(). By the soundness of the VIF, this change modifies the adversary's advantage by at most a negligible amount. By VIF pseudorandomness, we can then replace the per-identity public keys by fresh public keys output by ECDSA.KeyGen().

Finally, we replace the signatures produced during authentication with signatures generated using ECDSA.Sign(). The exfiltration resistance property of the firewalled signature scheme ensures that this change increases the adversary's success probability by at most a negligible amount. At this point, all the values sent to the relying party are sampled from the ideal distribution, as required.

Property 2: True2F protects against a compromised browser.
Proof idea: To prove the claim, we first replace the master secret key msk given to the honest token with one generated by VIF.KeyGen(). The bias-freeness of the key-generation protocol permits this change.

The zero knowledge property of the firewalled signature scheme implies that we can replace the signing protocol between the honest token and corrupt browser run during authentication (Step III.3 in Figure 6) by having the honest token send a properly generated signature σ to the browser. The remaining protocol transcript elements can be generated by the zero-knowledge simulator.

Then, protection against a compromised browser follows immediately from the VIF's unforgeability property. That is, to authenticate to a previously registered website without interacting with the token, the corrupt browser must produce a valid signature on a challenge, under $\mathsf{pk}_{\mathsf{id}}$, for some identity id, without querying the token for a signature on this challenge. (Here, we rely on the fact that before the browser is compromised, it samples the identities id from a space large enough to avoid collisions.)

Property 3: True2F protects against token fingerprinting.
Proof idea; First, we replace the adversarial token with an ideal token. By the "Protects against malicious token" property (above), this change cannot improve the adversary's success probability by a non-negligible amount.

Next, we replace our counter construction with $I = 100$ independent counters. The properties of our counter design ensure that this change does not increase the adversary's advantage, as long as the adversary performs no more than $A = 6.4$ million authentications.

Finally, we replace—one at a time—the per-identity public keys $\mathsf{pk}_{\mathsf{id}}$ generated by the VIF with independent random public keys. With each replacement, the pseudorandomness of the VIF ensures that the relying party cannot notice the change. At this point, the adversary's view in the two worlds is identical, so it has no hope of distinguishing them.

7 Implementation

The source code for our implementation is available at `https://github.com/edauterman/u2f-ref-code` and `https://github.com/edauterman/true2f`.

7.1 System architecture

Our True2F implementation consists of: (i) an extension to the Chrome browser, (ii) a local agent that runs alongside the browser, and (iii) a hardware token.

Browser Extension: Our browser extension forwards U2F requests to a local agent process that runs True2F with the token, and is compatible with relying parties that support U2F. We built on an existing extension for U2F development [65]. To do so, we wrote roughly 50 lines of JavaScript code. Our extension does not yet sync token state via a user's browser profile (Section 3.3).

Local Agent: The local agent runs the True2F protocol with the token. We built the local agent on top of an existing U2F reference implementation [65] using the OpenSSL library. To do so, we wrote roughly 3,800 lines of C/C++ code: 2,000 for running the cryptographic protocol, 700 for the simulating the token's flash-friendly counter (to enforce correct counter values), 800 for parsing and encoding, and 300 for the interface to the extension.

Token: We implemented the True2F protocol in firmware on a standard U2F hardware token used by Google employees. It uses an ARM SC-300 processor clocked at 24 MHz and has a cryptographic accelerator with an interface exposed for certain operations (see Table 7). Our implementation on the token required adding or modifying roughly 2,000 lines of C code: 1,400 for the protocol, and 600 for the counter.

Table 7: Cost of various operations on the token, averaged over 100 runs, and the expected number of each operation required per authentication attempt. "HW?" indicates use of the token's crypto accelerator.

Operation	HW?	Time (µs)	Ops. per auth.					
			No opts.	+ Fast keygen	+ VRF caching	+ Hash assist	**True2F** (+ all)	U2F
SHA256 (128 bytes)	Y	19	5	5	3	5	3	1
$x + y \in \mathbb{Z}_q$	N	36	17	16	2	15	1	0
$x \cdot y \in \mathbb{Z}_q$	N	409	9	8	2	11	1	0
$g^x \in \mathbb{G}$	Y	17,400	7	5	3	7	1	0
ECDSA.Sign	Y	18,600	1	1	1	1	1	1
$g \cdot h \in \mathbb{G}$	N	25,636	1	0	1	1	0	0
$\sqrt{x} \in \mathbb{Z}_q$	N	105,488	2	2	0	0	0	0

7.2 Cryptographic optimizations

We have implemented a number of optimizations to minimize the cost of the True2F protocols, and we sketch these here.

Browser-assisted hash-to-point. To evaluate the VRF used in our VIF construction, the token must compute a hash function $H_{\mathbb{G}}$ that maps bitstrings into points on the P256 elliptic curve. We implement $H_{\mathbb{G}}$ using the "try-and-increment" method [23, Section 3.3]. If P256 is defined over the field $\mathbb{F}_p$, the try-and-increment method requires the token to compute the square root modulo p of the *first* quadratic residue (i.e., square modulo p) in a sequence of values $Z = z_1, z_2, z_3, \cdots \in \mathbb{F}_p$. As Table 7 shows, computing square roots on the token is costly—more than 5× slower than computing signatures. (There are other methods for implementing $H_{\mathbb{G}}$, but these either do not apply to the P256 curve or also require computing square roots in $\mathbb{F}_p$ [15, 26, 70, 114].)

The token can outsource most of the $H_{\mathbb{G}}$ computation to the browser. To do so, we take advantage of the fact that when $\mathbb{F}_p$ is the P256 base field, if $z \in \mathbb{F}_p$ is a quadratic non-residue, then $-z \in \mathbb{F}_p$ is a quadratic residue. The outsourcing then works as follows: the token and browser both compute the sequence Z. Let z_ℓ be the first quadratic residue in Z. Then all values $z_1, \ldots, z_{\ell-1}$ are quadratic non-residues. For each such value z_i, the browser sends the token the square root r_i of $-z_i \in \mathbb{F}_p$. The token checks that $r_i^2 = -z_i \in \mathbb{F}_p$, for all $i \in \{1, \ldots, \ell-1\}$. Finally, the browser sends the square root r_ℓ of z_ℓ and the token checks that $r_\ell^2 = z_\ell \in \mathbb{F}_p$. Since *checking* a square-root computation (i.e., computing $x^2 \bmod p$) is $\approx 256\times$ faster that *computing* a square root modulo p, this outsourcing is profitable for the token.

This optimization brings the cost for the hash-to-point operation down to 2.5ms, from an unoptimized cost of 199ms.

Caching VRF outputs. When registering or authenticating with a site with identity id, the token uses our VIF construction (Section 4.2) to compute the per-identity signing key $\mathsf{sk}_{\mathsf{id}}$. To compute $\mathsf{sk}_{\mathsf{id}}$, the token must evaluate the VRF at the point id to get a VRF output $y_{\mathsf{id}} \in \mathbb{Z}_q$.

Even with the hash-to-point optimization described above, evaluating the VRF is relatively costly, since it requires computing an exponentiation in $\mathbb{G}$. We eliminate the need for the costly VRF computation during the authentication phase entirely by

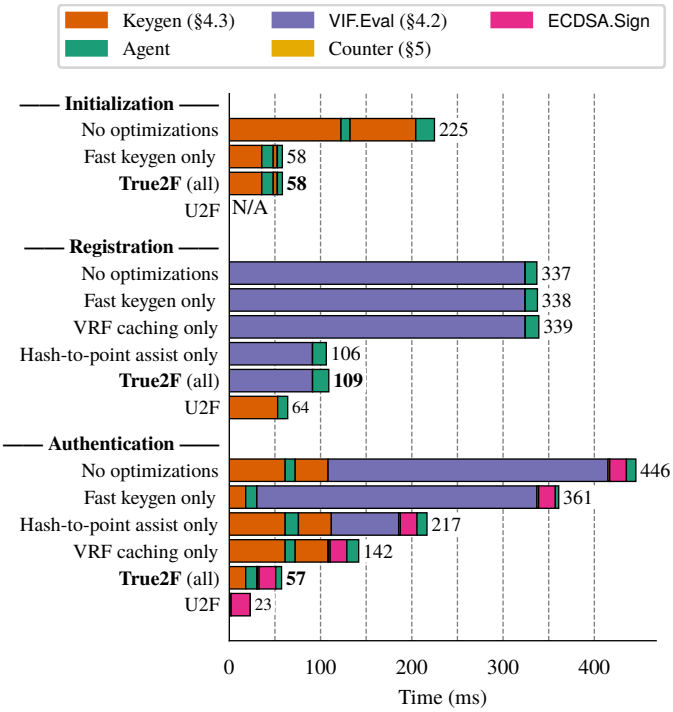

Figure 8: Protocol execution time for True2F, U2F, and less optimized variants of True2F, averaged across 100 runs. For all measurements, we instrument the token to not wait for user touch. The standard deviation for all measurements is less than 2ms.

computing a message authentication code (MAC) [10] and using the browser as an off-token cache. During initialization, the token generates and stores a MAC secret key. The MAC is only relevant when the token is honest, so the token can generate this MAC key on its own.

During registration, we have the token first compute the VRF output y_{id}. Then, the token computes a MAC tag τ_{id} over the tuple $\langle \mathsf{id}, y_{\mathsf{id}} \rangle$ and sends the triple $\langle \mathsf{id}, y_{\mathsf{id}}, \tau_{\mathsf{id}} \rangle$ to the browser, along with its registration response message. The MAC tag never leaves the browser, so the token cannot use it to exfiltrate data.

Later on, when the browser wants the token to authenticate to site id, the browser sends the token the triple $\langle \mathsf{id}, y_{\mathsf{id}}, \tau_{\mathsf{id}} \rangle$ along with its authentication request. After verifying the MAC tag, the token can use y_{id} to generate the per-identity secret key $\mathsf{sk}_{\mathsf{id}}$ without having to recompute the VRF at the point id.

Since computing and verifying a MAC just requires a few invocations of SHA256, this optimization brings the cost of evaluating the VIF down to 0.47ms, compared with 73.75ms when using only the hash-to-point optimization.

8 Evaluation

We evaluate True2F on the hardware token described in Section 7 and an agent running on an Intel Xeon W-2135 processor at 3.8GHz. Our True2F implementation uses 85KB bytes of token flash space total: 75KB of code and 10KB for keys and counters. For comparison, a plain U2F implementation uses 70KB of space: 64KB of code and 6KB of keys and counters.

Protocol execution time. Figure 8 gives the time to execute the initialization, registration, and authentication protocols for True2F, U2F, and partially optimized variants of True2F, timed

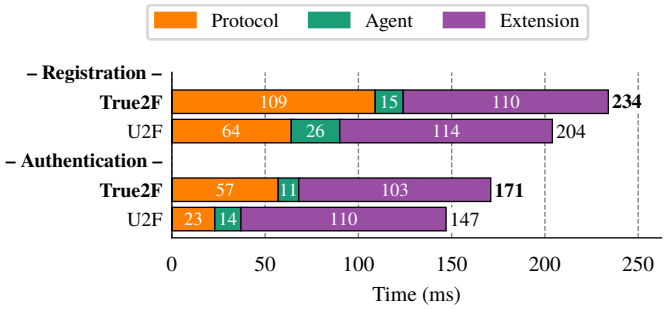

Figure 9: Total time to complete a FIDO U2F request.

from when the agent begins the protocol with the token to when the agent is ready to respond to the browser extension.

A True2F registration (109ms) is 1.7× slower than a U2F registration (64ms). Figure 8 demonstrates the effect of the hash-to-point optimization (Section 7.2) during registration—this optimization cuts execution time by over 230ms. A True2F authentication takes 57ms, which is 2.3× slower than an unprotected U2F authentication (23 ms). The cost of g^x computations dominates the authentication cost: our protocol requires four (two on the token and two on the agent) while unprotected U2F requires only one, to generate the ECDSA signature. Our VRF caching optimization (Section 7.2) essentially eliminates the cost of generating the per-identity signing key using VIF.Eval.

End-to-end latency. Figure 9 shows the total time for registering or authenticating with True2F and unprotected U2F, measured from the time that the webpage invokes the Javascript API to the time the API call returns. As Figure 9 shows, the total time for True2F authentication is 171ms, compared with 147ms for unprotected U2F, which amounts to a 16% slowdown.

Counters. Figure 10 shows the time required to increment a counter when using our log-structured counter design. When there is no need to garbage collect the log, an increment takes between 1.63ms and 2.35ms. The difference in increment time is a function of the log size: to look up the counter value associated with a relying party, the token must read through the entire log. To make increments constant time, a hardened implementation would read through the entire log flash page—not just to the end of the log. To avoid triggering an expensive garbage-collection operation during registration or authentication, we can execute garbage collection asynchronously (e.g., while the token is plugged in to a USB port but is idle).

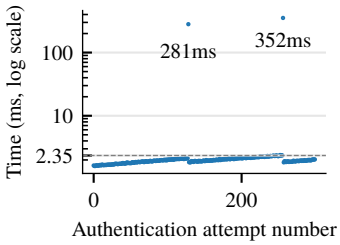

Figure 10: Counter update time with synchronous garbage collection.

The simple global counter design requires 0.43ms to increment, compared with a worst-case time of 2.35ms for our design. In absolute terms, the costly cryptographic operations we execute in True2F mask this 2ms counter-update time.

Browser sync state. To support using the same token with many browsers, True2F requires syncing state across multiple browser instances with the same user profile. If a user registers the token with I identities, an upper bound on the size of the sync state

is $4162 + 97 \cdot I$ bytes. For the fixed state, 4KB is needed to store the counters, and 66 bytes are needed for the VIF master public key. Each additional identity requires syncing a 32-byte MAC, a 32-byte VRF output, and a 33-byte public key. With 100 registered identities, this state amounts to less than 14KB, which is far less than the 100KB of state that the Chrome browser will sync per extension [33].

9 Related work

The threat of hardware backdoors ("hardware Trojans") motivates our work [111, 125]. Using dopant-level Trojans [9, 77], it is even possible to make a Trojan circuit look physically the same as a Trojan-free circuit.

There are three standard defenses against hardware Trojans [17]: (1) detect Trojans with logic testing or by comparing chip side-channel information (e.g. power and temperature) to that of a reference chip [1, 8, 72, 92, 101], (2) monitor runtime behavior using sensors to detect dramatic changes in the chip's behavior (e.g. signal propagation delay) [55, 80], and (3) design the hardware to make it more difficult to insert Trojans and easier to detect them [30, 71, 103, 110, 117, 123]. Cryptographic techniques, such as verifiable computation protocols, can audit a chip's behavior [7, 13, 18, 59, 63, 75, 100, 116, 118].

A backdoored U2F token could exfiltrate a user's cryptographic secrets by encoding them in innocent-looking protocol messages. Simmons introduced the notion of subliminal channels [108] to capture this class of attacks and Desmedt proposed "subliminal-free signatures" [44] as a defense. Later work extended the notion of subliminal-freeness to zero-knowledge protocols and other primitives [20, 28, 43, 45, 97, 108, 109].

Work on kleptography [104, 105, 127] and algorithm-substitution attacks [11, 54] models and defends against more general malicious implementations of cryptographic primitives.

Cryptographic reverse firewalls [87] are a general technique for defending against backdoored implementations of interactive cryptographic protocols. A reverse firewall sits between a potentially backdoored implementation and the outside world, modifying messages in such a way that (a) preserves the security of the original protocol and (b) prevents the malicious implementation from leaking information to the outside world. In this work, we essentially build a reverse firewall for the special case of U2F tokens.

Ateniese et al. show that if a digital signature scheme Σ is unique or rerandomizable then there are very efficient reverse firewalls for Σ [6]. Our firewalled signing protocol is a reverse firewall for ECDSA, which is neither unique nor rerandomizable. Our signing protocol, however, is interactive, while theirs is not.

Recent work explores two-party protocols for ECDSA signing when each party holds a share of the signing key [48, 83]. In contrast, in our setting the token has the entire secret key and the browser's role is to enforce that the token samples its signing randomness from the correct distribution. The problem of enforcing good use of randomness also appears in prior work on collaborative key-generation protocols [36, 66, 73].

True2F uses a log-structured counter design. Log-structured file systems go back to the work of Rosenblum and Ousterhout [102], and similar ideas appear in flash-oriented filesystems [39, 56, 120, 124], key-value stores [4, 41, 42, 82], and other data structures [3, 81, 85, 93, 121, 132], many of which are tailored to embedded devices. While much of this work focuses on building general file systems on flash, we seek extreme space efficiency by tailoring our design to the very specific needs of increment-only counters.

With the rise of browser protections against cookie abuses, advertisers and trackers turned to novel fingerprinting techniques, such as the installation of browser plugins [50, 95, 96]. U2F token fingerprinting gives advertisers yet another way to track users across origins [35, 88]. The defenses we introduce in True2F reduce the fingerprinting surface introduced by U2F.

10 Conclusions

True2F shows that it *is* possible to implement very strong defenses against hardware Trojans for one important class of hardware devices. Furthermore, the True2F protections maintain server-side backwards compatibility and come with unnoticeable performance overhead.

In future work, we hope to extend the True2F token design to handle post-quantum signature schemes—such as those based on lattices or hash functions [2, 16, 24, 31, 49]. A second task is to add a notion of third-party auditability to True2F. That is, if the browser ever outputs "Token failure," it should also output a third-party verifiable proof of the token's misbehavior. An honest token should be able to generate a similar proof if the browser ever misbehaves. Is it possible to achieve this stronger notion of security without sacrificing performance?

Acknowledgements. We would like to thank Bryan Ford and Philipp Jovanovic for their thoughts on applications and extensions of True2F. Phil Levis pointed us to relevant related work on data structures optimized for flash-storage. Saba Eskandarian and Dmitry Kogan gave comments that improved the writing. Marius Schilder provided implementation guidance and helped us understand flash hardware constraints.

This work received support from NSF, DARPA, ONR, the Simons Foundation, and CISPA. Opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. *Security and Privacy*, 2007.

[2] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange–a new hope. In *USENIX Security Symposium*, 2016.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *NSDI*, 2010.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[5] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals with cinch. In *USENIX Security Symposium*, Austin, TX, 2016.

[6] G. Ateniese, B. Magri, and D. Venturi. Subversion-resilient signatures: Definitions, constructions and applications. In *CCS*, 2015.

[7] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *Security and Privacy*. IEEE, 2015.

[8] J. Balasch, B. Gierlichs, and I. Verbauwhede. Electromagnetic circuit fingerprints for hardware trojan detection. In *Symposium on Electromagnetic Compatibility*. IEEE, 2015.

[9] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson. Stealthy dopant-level hardware trojans. In *CHES*. Springer, 2013.

[10] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.

[11] M. Bellare, K. G. Paterson, and P. Rogaway. Security of symmetric encryption against mass surveillance. In *CRYPTO*, 2014.

[12] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, 1993.

[13] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.

[14] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), 2012.

[15] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *CCS*, 2013.

[16] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *EUROCRYPT*, 2015.

[17] S. Bhunia, M. Hsiao, M. Banga, and S. Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8), 2014.

[18] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

[19] Hardware wallets. `https://en.bitcoin.it/wiki/Hardware_wallet`, Accessed 27 September 2018.

[20] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, 1998.

[21] J.-M. Bohli, M. I. G. Vasco, and R. Steinwandt. A subliminal-free variant of ecdsa. In *International Workshop on Information Hiding*, 2006.

[22] D. Boneh. The decision diffie-hellman problem. In *ANTS*, 1998.

[23] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of cryptology*, 17(4), 2004.

[24] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *CCS*, 2016.

[25] E. Brickell, D. Pointcheval, S. Vaudenay, and M. Yung. Design validations for discrete logarithm based signature schemes. In *PKC*, 2000.

[26] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *CRYPTO*, 2010.

[27] D. R. Brown. Generic groups, collision resistance, and ECDSA. *Designs, Codes and Cryptography*, 35(1), 2005.

[28] M. Burmester, Y. Desmedt, T. Itoh, K. Sakurai, H. Shizuya, and M. Yung. A progress report on subliminal-free channels. In *Information Hiding*, 1996.

[29] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: design and detection. In *CCS*, 2004.

[30] R. S. Chakraborty and S. Bhunia. HARPOON: An obfuscation-based SoC design methodology for hardware protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28:1493–1502, 2009.

[31] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. *Report on post-quantum cryptography*. NIST, 2016.

[32] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security Symposium*, 2005.

[33] chrome.storage api documentation. `https://developer.chrome.com/extensions/storage`.

[34] Security keys. `https://www.chromium.org/security-keys`.

[35] A. Cooper, H. Tschofenig, B. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith. Privacy considerations for internet protocols. RFC 6973, RFC Editor, July 2013.

[36] H. Corrigan-Gibbs, W. Mu, D. Boneh, and B. Ford. Ensuring high-quality randomness in cryptographic key generation. In *CCS*, 2013.

[37] J. Cox. Experts call for transparency around Google's Chinese-made security keys, August 31, 2018. `https://motherboard.vice.com/en_us/article/mb4zy3/transparency-google-titan-security-keys-china`.

[38] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *HotSec*, 2008.

[39] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured flash file system for micro sensor nodes. In *SenSys*, 2004.

[40] I. B. Damgård, T. P. Pedersen, and B. Pfitzmann. On the existence of statistically hiding bit commitment schemes and fail-stop signatures. *Journal of Cryptology*, 10(3), 1997.

[41] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, 2010.

[42] B. K. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *SIGMOD*, 2011.

[43] Y. Desmedt. Abuses in cryptography and how to fight them. In *CRYPTO*, New York, NY, 1988. Springer New York.

[44] Y. Desmedt. Subliminal-free authentication and signature. In *EUROCRYPT*, 1988.

[45] Y. Desmedt. Subliminal-free sharing schemes. In *International Symposium on Information Theory*. IEEE, 1994.

[46] Y. Dodis, I. Mironov, and N. Stephens-Davidowitz. Message transmission with reverse firewalls—secure communication on corrupted machines. In *CRYPTO*, 2016.

[47] Y. Dodis and A. Yampolskiy. A verifiable random function with short proofs and keys. In *PKC*, 2005.

[48] J. Doerner, Y. Kondi, E. Lee, and a. shelat. Secure two-party threshold ecdsa from ecdsa assumptions. In *Security and Privacy*. IEEE, 2018.

[49] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS–Dilithium: a lattice-based digital signature scheme. In *CHES*, number 1, 2018.

[50] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *CCS*, 2016.

[51] A. Everspaugh, Y. Zhai, R. Jellinek, T. Ristenpart, and M. Swift. Not-so-random numbers in virtualized Linux and the Whirlwind RNG. In *Security and Privacy*. IEEE, 2014.

[52] M. Fersch, E. Kiltz, and B. Poettering. On the provable security of (EC)DSA signatures. In *CCS*, 2016.

[53] FIDO Alliance. FIDO U2F raw message formats, Apr. 2017. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html.

[54] M. Fischlin and S. Mazaheri. Self-guarding cryptographic protocols against algorithm substitution attacks. In *Computer Security Foundations Symposium*. IEEE, July 2018.

[55] D. Forte, C. Bao, and A. Srivastava. Temperature tracking: An innovative run-time approach for hardware trojan detection. In *Conference on Computer-Aided Design*, 2013.

[56] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *ATEC*, pages 7–7, 2005.

[57] S. Gallagher. Photos of an NSA "upgrade" factory show Cisco router getting implant, May 14, 2014. `https://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/`.

[58] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *ACM SIGOPS European Workshop*, 2004.

[59] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[60] S. Goldberg, L. Reyzin, D. Papadopoulos, and J. Vcelak. Verifiable random functions (VRFs). IETF CFRG Internet-Draft (Standards Track), Mar. 2018. `https://tools.ietf.org/html/draft-irtf-cfrg-vrf-01`.

[61] O. Goldreich. *Foundations of cryptography*, volume 1. Cambridge University Press, 2001.

[62] O. Goldreich, S. Goldwasser, and S. Micali. How to construct randolli functions. In *FOCS*. IEEE, 1984.

[63] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.

[64] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2), 1988.

[65] Google. U2F reference implementations.

[66] L. Hanzlik, K. Kluczniak, and M. Kutyłowski. Controlled randomness–a defense against backdoors in cryptographic devices. In *International Conference on Cryptology in Malaysia*, pages 215–232. Springer, 2016.

[67] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, volume 8, page 1, 2012.

[68] J. Hong. The state of phishing attacks. *Communications of the ACM*, 55(1):74–81, 2012.

[69] W.-M. Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[70] T. Icart. How to hash into elliptic curves. In *CRYPTO*. Springer, 2009.

[71] F. Imeson, A. Emtenan, S. Garg, and M. V. Tripunitara. Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In *USENIX Security Symposium*, 2013.

[72] Y. Jin and Y. Makris. Hardware trojan detection using path delay fingerprint. In *Workshop on Hardware-Oriented Security and Trust*. IEEE,

2008.
[73] A. Juels and J. Guajardo. RSA key generation with verifiable randomness. In *PKC*, 2002.
[74] D. Kohlbrenner and H. Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, 2016.
[75] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. Trueset: Faster verifiable set computations. In *USENIX Security Symposium*, 2014.
[76] B. Krebs. Google: Security keys neutralized employee phishing, July 18, 2018. https://krebsonsecurity.com/2018/07/google-security-keys-neutralized-employee-phishing/.
[77] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric trojans for fault-injection attacks on cryptographic hardware. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept. 2014.
[78] J. Lang, A. Czeskis, D. Balfanz, and M. Schilder. Security keys: Practical cryptographic second factors for the modern web. In *Financial Cryptography*, 2016.
[79] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012.
[80] J. Li and J. Lach. At-speed delay characterization for IC authentication and trojan horse detection. In *Workshop on Hardware-Oriented Security and Trust*, 2008.
[81] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *VLDB*, 3(1-2), 2010.
[82] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *SOSP*, 2011.
[83] Y. Lindell. Fast secure two-party ECDSA signing. In *CRYPTO*, 2017.
[84] Y. Lindell and J. Katz. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.
[85] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys*, pages 195–208. ACM, 2006.
[86] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *FOCS*, 1999.
[87] I. Mironov and N. Stephens-Davidowitz. Cryptographic reverse firewalls. In *EUROCRYPT*, 2015.
[88] Mitigating browser fingerprinting in web specifications, July 2018. http://w3c.github.io/fingerprinting-guidance/.
[89] H. Morita, J. C. Schuldt, T. Matsuda, G. Hanaoka, and T. Iwata. On the security of the Schnorr signature scheme and DSA against related-key attacks. In *International Conference on Information Security and Cryptology*, 2015.
[90] M. Naor, R. Ostrovsky, R. Venkatesan, and M. Yung. Perfect zero-knowledge arguments for NP using any one-way permutation. *Journal of Cryptology*, 11(2), 1998.
[91] M. Naor and A. Ziv. Primary-secondary-resolver membership proof systems. In *TCC*, 2015.
[92] S. Narasimhan, X. Wang, D. Du, R. S. Chakraborty, and S. Bhunia. TeSR: A robust temporal self-referencing approach for hardware Trojan detection. In *Symposium on Hardware-Oriented Security and Trust*, 2011.
[93] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for nand flash. In *IPSN*, 2007.
[94] M. Nemec, M. Sys, P. Svenda, D. Klinec, and V. Matyas. The return of Coppersmith's Attack: Practical factorization of widely used RSA moduli. In *CCS*, 2017.
[95] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. On the workings and current practices of web-based device fingerprinting. *IEEE Security and Privacy*, 12(3), May-June 2014.
[96] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Krügel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. *Security and Privacy*, 2013.
[97] T. Okamoto and K. Ohta. Divertible zero knowledge interactive proofs and commutative random self-reducibility. In *EUROCRYPT*, 1989.
[98] C. Osborne. Why is Google selling potentially compromised chinese security keys?, August 31 2018. https://www.zdnet.com/article/google-launches-titan-security-keys-but-recommends-keys-from-chinese-firm-with-military-links-in/.
[99] D. Papadopoulos, D. Wessels, S. Huque, M. Naor, J. Včelák, L. Reyzin, and S. Goldberg. Making NSEC5 practical for DNSSEC. Cryptology ePrint Archive, Report 2017/099, 2017.
[100] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy*, volume 59. IEEE, 05 2013.
[101] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey. Hardware trojan horse detection using gate-level characterization. In *Design Automation Conference*, 2009.
[102] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1), February 1992.
[103] J. A. Roy, F. Koushanfar, and I. L. Markov. EPIC: Ending piracy of integrated circuits. In *Design, Automation and Test in Europe*, March 2008.
[104] A. Russell, Q. Tang, M. Yung, and H.-S. Zhou. Cliptography: Clipping the power of kleptographic attacks. In *ASIACRYPT*. Springer, 2016.
[105] A. Russell, Q. Tang, M. Yung, and H.-S. Zhou. Generic semantic security against a kleptographic adversary. In *CCS*, CCS '17, 2017.
[106] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3), 1991.
[107] K. Sedgwick. Man's life savings stolen from hardware wallet supplied by a reseller, January 6,, 2018. https://news.bitcoin.com/mans-life-savings-stolen-from-hardware-wallet-supplied-by-a-reseller/.
[108] G. J. Simmons. The Prisoners' Problem and the Subliminal Channel. In *CRYPTO*, 1984.
[109] G. J. Simmons. The subliminal channel and digital signatures. In *EUROCRYPT*, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
[110] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building stealthy and malicious hardware. In *Security and Privacy*. IEEE, 2011.
[111] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design Test of Computers*, 27(1), Jan 2010.
[112] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB great again with USBFILTER. In *USENIX Security Symposium*, 2016.
[113] J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. Butler. SoK: "plug & pray" today - understanding USB insecurity in versions 1 through c. In *Security and Privacy*. IEEE, 2018.
[114] M. Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In *Financial Cryptography*, 2014.
[115] W3C. FIDO Alliance and W3C achieve major standards milestone in global effort towards simpler, stronger authentication on the web, 2018.
[116] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish. Verifiable ASICs. In *Security and Privacy*. IEEE, 2016.
[117] A. Waksman, M. Suozzo, and S. Sethumadhavan. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *CCS*, 2013.
[118] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), Jan. 2015.
[119] Web authentication. https://www.w3.org/TR/2018/CR-webauthn-20180320/.
[120] D. Woodhouse. JFFS: The journalling flash file system. In *Ottawa Linux Symposium*, 2001.
[121] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *TECS*, 2007.
[122] P. Wuille. Hierarchical deterministic wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, 2 2012. Bitcoin Improvement Proposal #32.
[123] K. Xiao, D. Forte, and M. M. Tehranipoor. A novel built-in self-authentication technique to prevent inserting hardware trojans. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33:1778–1791, 2014.
[124] Yaffs. https://yaffs.net/.
[125] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog malicious hardware. In *Security and Privacy*. IEEE, May 2016.
[126] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In *IMC*, 2009.
[127] A. Young and M. Yung. Kleptography: Using cryptography against cryptography. In *EUROCRYPT*, 1997.
[128] Key generation. https://developers.yubico.com/U2F/Protocol_details/Key_generation.html.
[129] Yubico. FIDO U2F, 2018. https://www.yubico.com/solutions/fido-u2f/.
[130] Yubico. Security advisory 2018-03-02 - WebUSB bypass of U2F phishing protection, 3 2018. https://www.yubico.com/support/security-advisories/ysa-2018-02/.
[131] Yubico. What is FIDO2?, 2018. https://developers.yubico.com/FIDO2/.
[132] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *FAST*, 2005.

A Selective failure attacks

In our security definitions (Section 3.2), we restrict our attention to tokens that never cause an honest browser to abort. We explain here why this restriction is not too severe.

Informally, say that an adversarial relying party $\mathcal{A}_{\mathsf{rp}}$ wants to collude with a malicious token $\mathcal{A}_{\mathsf{t}}$ to achieve some goal $\mathcal{G}$. For example, a malicious token might want to leak its cryptographic signing key to the malicious relying party. We claim that if the adversary $(\mathcal{A}_{\mathsf{rp}}, \mathcal{A}_{\mathsf{t}})$ achieves goal $\mathcal{G}$ with probability at most ϵ when interacting with an adversarial token that *never* causes the honest browser to abort, then the adversarial relying party achieves goal $\mathcal{G}$ with probability at most $\epsilon_{\mathsf{abort}} \leq (T+1)\epsilon$, after at most T registration or authentication interactions with a token that possibly causes an abort at a chosen time.

Thus, if we can show that the probability of achieving $\mathcal{G}$ is negligible when the token never causes the honest browser to abort, then this probability remains negligible even if the token triggers an abort at an arbitrary time. So a malicious token cannot use selective failure to exfiltrate a cryptographic key, unless such an attack were possible without selective failure.

To prove the claim informally: Assume that there exists an adversarial relying party and token $\mathcal{A} = (\mathcal{A}_{\mathsf{rp}}, \mathcal{A}_{\mathsf{t}})$ such that (a) $\mathcal{A}_{\mathsf{t}}$ possibly triggers a browser abort and (b) $\mathcal{A}$ achieves goal $\mathcal{G}$ with probability $\epsilon_{\mathsf{abort}}$. Then we we can construct an adversary $\mathcal{A}' = (\mathcal{A}'_{\mathsf{t}}, \mathcal{A}'_{\mathsf{rp}})$ such that (a) $\mathcal{A}'_{\mathsf{t}}$ never triggers a browser abort and (b) $\mathcal{A}'$ achieves goal $\mathcal{G}$ with probability $\epsilon = \epsilon_{\mathsf{abort}}/(T+1)$.

The token $\mathcal{A}'_{\mathsf{t}}$ runs $\mathcal{A}_{\mathsf{t}}$, except that on the request for which $\mathcal{A}_{\mathsf{t}}$ would have triggered a browser abort, $\mathcal{A}'_{\mathsf{t}}$ executes the protocol faithfully. The relying party $\mathcal{A}'_{\mathsf{rp}}$ just guesses the index of the interaction on which $\mathcal{A}_{\mathsf{t}}$ would have caused an abort (or that $\mathcal{A}_{\mathsf{t}}$ never causes an abort). This guess is correct with independent probability $1/(T+1)$, so the advantage of $\mathcal{A}'$ is at least $\epsilon = \epsilon_{\mathsf{abort}}/(T+1)$.

This argument crucially relies on the fact that once a token fails, the user discards the token (i.e., the token can only abort *once*), so it is important that the browser interface prevent a user from continuing to use a failing token.

B The ECDSA signature scheme

We follow the concise description of ECDSA of Fersch, Kiltz, and Poettering [52]. The ECDSA signature scheme over message space $\mathcal{M}$ uses a fixed group $\mathbb{G} = \langle g \rangle$ of prime order q. For our applications, $\mathbb{G}$ is the NIST P256 elliptic curve group. The scheme also uses a hash function $H : \mathcal{M} \to \mathbb{Z}_q$ and a "conversion function" $f : \mathbb{G} \to \mathbb{Z}_q$.

The algorithms of the signature scheme are:

- $\mathsf{ECDSA.KeyGen}() \to (\mathsf{sk}, \mathsf{pk})$. Sample $x \xleftarrow{R} \mathbb{Z}_q$. Output x as the secret key and $X = g^x \in \mathbb{G}$ as the public key.
- $\mathsf{ECDSA.Sign}(\mathsf{sk} = x, m) \to \sigma$.
  - Choose $r \xleftarrow{R} \mathbb{Z}_q^*$.
  - Compute $c \leftarrow f(g^r) \in \mathbb{Z}_q$.
  - Compute $s \leftarrow (H(m) + c \cdot x)/r \in \mathbb{Z}_q$.
  - Output $\sigma = (c, s)$.

  We write $\mathsf{ECDSA.Sign}(\mathsf{sk}, m; r)$ to denote the deterministic operation of running $\mathsf{ECDSA.Sign}$ using randomness $r \in \mathbb{Z}_q^*$.
- $\mathsf{ECDSA.Verify}(\mathsf{pk}, m, \sigma) \to \{0, 1\}$.
  - Parse the verification key pk as a group element $X \in \mathbb{G}$. Parse the signature σ as a pair $(c, s) \in \mathbb{Z}_q^2$.
  - Compute the value
  $$R_{\text{abs}} \leftarrow (g^{H(m)} X^c)^{1/s} \quad \in \mathbb{G}. \tag{1}$$
  - Output "1" iff $f(R_{\text{abs}}) = c$.

In practice, H is a cryptographic hash function (e.g., SHA) and f is the function that interprets a group element $g \in \mathbb{G}$ as an elliptic-curve point and outputs the value of the x-coordinate of this point modulo q.

ECDSA signatures are malleable. ECDSA signatures are *malleable*: that is, given a valid signature σ on m under public key X, anyone can produce another valid signature $\bar{\sigma} \neq \sigma$ on the same message m under the same public key X. The reason is that if $\sigma = (c, s) \in \mathbb{Z}_q^2$ is a valid signature, then $\bar{\sigma} = (c, -s) \in \mathbb{Z}_q^2$ is also a valid signature.

Idealized ECDSA. We prove that the VIF construction of Section 4.2 has Σ-pseudorandomness and Σ-unforgeability when Σ is an "Idealized" variant of the ECDSA signature scheme. Following Brickell et al. [25], we idealize ECDSA in the sense that we model the two hash functions used in the ECDSA signature scheme as random oracles [12]. Modeling the conversion function f as a random oracle is somewhat problematic, as f satisfies a number of properties that a truly random function does not. For example, for any $X \in \mathbb{G}$, $f(X) = f(1/X)$ (using multiplicative notation for the group operation). Unfortunately, the peculiarities of ECDSA require resorting to some sort of idealization [27, 52].

C VIF security analysis

The experiments and algorithms in this section are are all implicitly parameterized by a security parameter and we require that all algorithms run in time polynomial in this parameter.

Standard definitions. We use the standard definition of *pseudorandomness* for a verifiable random function, as defined by Micali et al. [86]. We denote the VRF distinguishing advantage of an adversary $\mathcal{A}$ against a VRF $\mathcal{V}$ as $\mathsf{VRFAdv}[\mathcal{A}, \mathcal{V}]$. We use the standard definition of security of digital signature schemes: *existential unforgeability under chosen message attacks*. See Katz and Lindell [84] for a formal definition. We denote the forging probability of an adversary $\mathcal{A}$ against a signature scheme Σ as $\mathsf{SigAdv}[\mathcal{A}, \Sigma]$.

C.1 VIF security definitions

In this section, we give the complete definition of security for the *verifiable identity family* primitive introduced in Section 4.2.

Defining pseudorandomness. We define pseudorandomness using Experiment 1, shown in Figure 11. For a bit $b \in \{0, 1\}$, let $p_{1,b}$ denote the probability that the output of Experiment 1, with challenge bit b, outputs "1." Then define the advantage of an adversary $\mathcal{A}$ at attacking the pseudorandomness of a VRF scheme $\mathcal{V}$ as: $\mathsf{VIF{-}PRAdv}[\mathcal{A}, \Phi_\Sigma] \stackrel{\text{def}}{=} \left| p_{1,0} - p_{1,1} \right|$. A VIF scheme Φ_Σ satisfies Σ-*pseudorandomness* if for all efficient adversaries $\mathcal{A}$, $\mathsf{VIF{-}PRAdv}[\mathcal{A}, \Phi_\Sigma]$ is negligible in the security parameter.

Defining unforgeability. We define unforgeability using Experiment 2 shown in Figure 12. Define the Σ-Unforgeability advantage of an adversary $\mathcal{A}$ at attacking a VIF scheme Φ_Σ as the probability that Experiment outputs "1." Denote this probability

Experiment 1: VIF Σ-Pseudorandomness. The experiment is parameterized by a signature scheme $\Sigma = (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$ with message space $\mathcal{M}$, a VIF scheme $\Phi_\Sigma = (\mathsf{VIF.KeyGen}, \mathsf{VIF.Eval}, \mathsf{VIF.Verify})$ with identity space $\mathcal{I}$, a bit $b \in \{0,1\}$, and an adversary $\mathcal{A}$.
The experiment is an interaction between a challenger and the adversary $\mathcal{A}$, and it proceeds as follows:

1. The challenger runs $(\mathsf{msk}, \mathsf{mpk}) \leftarrow \mathsf{VIF.KeyGen}()$. and sends mpk to the adversary.
2. The challenger gives the adversary access to an **identity oracle** $\mathcal{O}_{\mathsf{id}}$ and a **signature oracle** $\mathcal{O}_{\mathsf{sig}}$, defined as follows:
   - $\mathcal{O}_{\mathsf{id}}(\mathsf{id} \in \mathcal{I})$:
     - $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}, \pi) \leftarrow \mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id})$
     - Return $(\mathsf{pk}_{\mathsf{id}}, \pi)$
   - $\mathcal{O}_{\mathsf{sig}}(\mathsf{id} \in \mathcal{I}, m \in \mathcal{M})$:
     - $(\mathsf{sk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id}}, \pi) \leftarrow \mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id})$
     - Return $\mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{id}}, m)$
3. At some point, the adversary $\mathcal{A}$ sends an identity id^* to the challenger. The adversary must not have previously queried the identity or signature oracles at id^*.
   - If $b = 0$, the challenger runs $(\mathsf{sk}_{\mathsf{id}^*}, \mathsf{pk}_{\mathsf{id}^*}, \pi) \leftarrow \mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id}^*)$, and sends $\mathsf{pk}_{\mathsf{id}^*}$ to the adversary.
   - If $b = 1$, the challenger runs $(\mathsf{sk}_{\mathsf{rand}}, \mathsf{pk}_{\mathsf{rand}}) \leftarrow \mathsf{Sig.KeyGen}()$, and sends $\mathsf{pk}_{\mathsf{rand}}$ to the adversary.
4. The adversary may continue to make identity- and signature-oracle queries, provided that the adversary never makes an identity query using identity id^*. The adversary may make subsequent signing queries for identity id^*:
   - If $b = 0$, the challenger signs using $\mathsf{sk}_{\mathsf{id}^*}$.
   - If $b = 1$, the challenger signs using $\mathsf{sk}_{\mathsf{rand}}$.
5. Finally, the adversary outputs a guess $\hat{b}$ for b, and the value $\hat{b}$ is the output of the experiment.

Figure 11: VIF pseudorandomness experiment.

as $\mathsf{VIF{-}UFAdv}[\mathcal{A}, \Phi_\Sigma]$. A VIF scheme Φ_Σ, defined relative to a signature scheme Σ, satisfies Σ-*unforgeability* if for all efficient adversaries $\mathcal{A}$, $\mathsf{VIF{-}UFAdv}[\mathcal{A}, \Phi_\Sigma]$ is negligible in the security parameter.

C.2 VIF security proofs

Theorem 4. *The VIF construction of Section 4.2, satisfies Σ-pseudorandomness when Σ is Idealized ECDSA, of Appendix B, and the VIF is instantiated with a secure VRF.*

Proof. From an adversary $\mathcal{A}$ that breaks the Σ-pseudorandomness of the VIF, we construct an adversary $\mathcal{B}$ that breaks the pseudorandomness of the VRF. The algorithm $\mathcal{B}$ works as follows:

- Receive a VRF public key $\mathsf{pk}_{\mathsf{VRF}}$ from the VRF challenger.
- Generate an ECDSA signing key $x \xleftarrow{R} \mathbb{Z}_q$, and the corresponding public key $X = g^x \in \mathbb{G}$. Then, send $(X, \mathsf{pk}_{\mathsf{VRF}})$ to $\mathcal{A}$ as the VIF master public key.
- Algorithm $\mathcal{A}$ makes identity and signing queries.
  - For each identity query id that $\mathcal{A}$ makes, query the VRF challenger on id and receive a response (y, π_{VRF}). Respond to the query with the pair $(\mathsf{pk}_{\mathsf{id}}, \pi)$, where $\mathsf{pk}_{\mathsf{id}} \leftarrow X^y$ and $\pi \leftarrow (y, \pi_{\mathsf{VRF}})$.

Experiment 2: VIF Σ-Unforgeability. The experiment is parameterized by a signature scheme Σ with message space $\mathcal{M}$, and a VIF scheme $\Phi_\Sigma = (\mathsf{VIF.KeyGen}, \mathsf{VIF.Eval}, \mathsf{VIF.Verify})$ with identity space $\mathcal{I}$, and an adversary $\mathcal{A}$.
The experiment is an interaction between a challenger and the adversary $\mathcal{A}$, and it proceeds as follows:

1. The challenger runs $(\mathsf{msk}, \mathsf{mpk}) \leftarrow \mathsf{VIF.KeyGen}()$. and sends mpk to the adversary.
2. The challenger gives the adversary access to an **identity oracle** and a **signature oracle**, as in Experiment 1.
3. The adversary $\mathcal{A}$ outputs a tuple $(\mathsf{id}^*, m^*, \sigma^*)$, where $\mathsf{id}^* \in \mathcal{I}$ $m^* \in \mathcal{M}$.
4. The output of the experiment is the bit $b = 1$ if
   - the adversary never queried the signature oracle on the pair (id^*, m^*), and
   - σ^* is a valid signature on m^* using the public key corresponding to identity id^*. That is, if $(\mathsf{sk}_{\mathsf{id}^*}, \mathsf{pk}_{\mathsf{id}^*}, \pi) \leftarrow \mathsf{VIF.Eval}(\mathsf{msk}, \mathsf{id}^*)$ and $\mathsf{Sig.Verify}(\mathsf{pk}_{\mathsf{id}^*}, m^*, \sigma^*) = 1$.

   The output is $b = 0$ otherwise.

Figure 12: VIF unforgeability experiment.

  - For each signing query (id, m) that $\mathcal{A}$ makes, query the VRF challenger on identity id, and receive a pair (y, π). Then, compute $\mathsf{sk}_{\mathsf{id}} \leftarrow x \cdot y \in \mathbb{Z}_q$ and $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{id}}, m)$, and return σ to $\mathcal{A}$.
- Upon receiving the VIF challenge point id^* from $\mathcal{A}$, forward this value to the VRF challenger. Receive the VRF challenge y^* from the VRF challenger, set $\mathsf{sk}_{\mathsf{id}^*} \leftarrow x \cdot y^* \in \mathbb{Z}_q$, and send $Y^* \leftarrow g^{x \cdot y^*} \in \mathbb{G}$ to the adversary $\mathcal{A}$.
  If $\mathcal{A}$ makes a signing query for message m on identity id^*, respond with $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{id}^*}, m)$.
- Output whatever $\mathcal{A}$ outputs.

When running $\mathcal{B}$ in the VRF experiment with VRF $\mathcal{V}$ and challenge bit b, $\mathcal{A}$'s view is exactly as in Experiment 1 with VRF $\mathcal{V}$ and challenge bit b. Thus, $\mathcal{B}$ achieves the same advantage at attacking the VRF $\mathcal{V}$ as $\mathcal{A}$ does in attacking the VIF's ECDSA-pseudorandomness property. □

Theorem 5. *The VIF construction of Section 4.2 satisfies Σ-unforgeability when Σ is the Idealized ECDSA signature scheme of Appendix B, when instantiated with a secure VRF, and when both H and f are modeled as random-oracles.*

The full version of the proof of Theorem 5 is straightforward, but quite lengthly. We give the proof idea here, and include the complete proof in the full version of the paper.[1]

Proof idea for Theorem 5. From an adversary $\mathcal{A}$ that breaks the Σ-unforgeability property of our VIF, we build a forger $\mathcal{B}_{\mathsf{sig}}$ for Idealized ECDSA. The reduction $\mathcal{B}_{\mathsf{sig}}$ takes as input an ECDSA public key $X \in \mathbb{G}$ and must produce a forged message-signature pair that verifies under X. The algorithm $\mathcal{B}_{\mathsf{sig}}$ first generates a fresh VRF keypair $(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{pk}_{\mathsf{VRF}}) \xleftarrow{R} \mathsf{VRF.KeyGen}()$. Then, $\mathcal{B}_{\mathsf{sig}}$ sends the pair $(X, \mathsf{pk}_{\mathsf{VRF}})$ to $\mathcal{A}$ as the VIF master public key.

The reduction must answer $\mathcal{A}$'s random-oracle queries (to H and f), identity queries, and signing queries.

[1] https://arxiv.org/abs/1810.04660

- To answer $\mathcal{A}$'s H-queries, $\mathcal{B}_{\mathsf{sig}}$ forwards them to its challenger.
- To answer $\mathcal{A}$'s f-queries, $\mathcal{B}_{\mathsf{sig}}$ lazily defines a fresh random function f', except as required to "program" f', as we describe below.
- To answer $\mathcal{A}$'s identity queries, $\mathcal{B}_{\mathsf{sig}}$ uses the VRF keypair $(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{pk}_{\mathsf{VRF}})$, as in the real construction.
- To answer $\mathcal{A}$'s signing queries, $\mathcal{B}_{\mathsf{sig}}$ "programs" the oracle f'. When the adversary makes a signing query (id, m), $\mathcal{B}_{\mathsf{sig}}$:
  - samples $c \xleftarrow{R} \mathbb{Z}_q$ and $s \xleftarrow{R} \mathbb{Z}_q^*$,
  - computes $y \leftarrow \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}) \in \mathbb{Z}_q^*$,
  - computes $R \leftarrow (g^{H(m)}(X^y)^c)^{1/s}$,
  - sets $f'(R) \leftarrow c \in \mathbb{Z}_q$, and
  - returns $\sigma = (c, s)$ to $\mathcal{A}$.

Eventually, the forger $\mathcal{A}$ outputs a triple $(\mathsf{id}^*, m^*, \sigma^*)$. Say that $X^* = X^{y^*} \in \mathbb{G}$ is the VIF public key associated with identity id^*. Then verifying the signature requires querying the function f' at the point $R^* = (g^{H(m^*)}(X^*)^{c^*})^{1/s^*}$, where $\sigma^* = (c^*, s^*) \in \mathbb{Z}_q^2$.

To make use of this forgery, the reduction $\mathcal{B}_{\mathsf{sig}}$ works by first (a) guessing the index of the identity query id^* on which $\mathcal{A}$ will forge and (b) guessing the index of the random oracle query to f' on which the adversary queries R^*.

If the algorithm $\mathcal{B}_{\mathsf{sig}}$ guesses both indices correctly, it can program the random oracle f' to satisfy $f'(R^*) = f(R^*)/y^*$. In this case, the pair $(f(R^*), s^*)$ is a valid signature on message m^* under ECDSA public key $X \in \mathbb{G}$, and we are done.

The tedium of the proof comes in making sure that, for example, the adversary actually makes an identity query at point id^* before querying f at the point R^*. (If $\mathcal{A}$ violates this ordering of events, then $\mathcal{B}_{\mathsf{sig}}$ will not know which value y^* to use to program the value of f' at the point R^*.) Proving this step is where we invoke the security of the VRF. We also must show, using a standard argument, that no two signatures use the same ECDSA signing nonce. If such collisions occur, we may not be able to program f' appropriately.

Putting all of these pieces together yields the theorem. □

D Security of key-generation protocol

We now analyze the protocol of Section 4.3.

Bias-free (honest browser). If the browser is honest, then the values v and r it chooses in Step 1 are distributed independently and uniformly over $\mathbb{Z}_q$. If the random oracle $H(\cdot,\cdot)$ yielded a perfectly hiding commitment scheme, then the value $V' \in \mathbb{G}$ that the token sends in Step 2 would be independent of v and thus the browser's output $X = V' \cdot g^v \in \mathbb{G}$ would be uniform over $\mathbb{G}$. Since the random oracle $H(\cdot,\cdot)$ only yields a statistically hiding commitment, the resulting distribution is instead statistically close to uniform.

Bias-free (honest token). If the token is honest, then the value $v' \in \mathbb{Z}_q$ it chooses in Step 2 of the protocol is distributed independently and uniformly at random over $\mathbb{Z}_q$. Once the browser commits to a pair $(v, r) \in \mathbb{Z}_q^2$ in Step 1, the probability that it can find a second pair $(v^*, r^*) \in \mathbb{Z}_q^2$ with $v \neq v^*$ such that $H(v, r) = H(v^*, r^*)$ is negligible. In other words, an efficient browser can only produce a single value v in Step 3, except with negligible probability. As long as this failure event does not occur, v is distributed independently of v' and $x = v + v' \in \mathbb{Z}_q$ is distributed uniformly over $\mathbb{Z}_q$.

Zero knowledge. Given an efficient adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ representing a malicious browser, we construct an efficient simulator Sim (at right) that takes as input a value $X \in \mathbb{G}$ and simulates the browser's view of the protocol. The simulator need only work for an adversary that never causes an honest token to output "$\perp$." To argue that the simulation is correct: Since the adversary $\mathcal{A}$ never causes the honest token to output "$\perp$," we know that $c = H(v, r) = H(v^*, r^*)$. The binding property of H (when viewed as a commitment scheme) implies that the probability that the simulation aborts due to the $v \neq v^*$ event, is negligible.

$\mathsf{Sim}(X)$:
$(\mathsf{st}, c) \leftarrow \mathcal{A}_1()$
$R \xleftarrow{R} \mathbb{G}$.
$(v, r) \leftarrow \mathcal{A}_2(\mathsf{st}, R)$.
$V' \leftarrow (X/g^v) \in \mathbb{G}$.
$(v^*, r^*) \leftarrow \mathcal{A}_2(\mathsf{st}, V')$.
If $v \neq v^*$, abort.
Else, return $(c, V', (v, r^*))$.

Given that the simulation does not abort, the adversary's view is simulated perfectly: the value V' satisfies $X = V' \cdot g^v$, as in the real interaction, and the other values are identical as well.

E Security analysis of our firewalled signatures

Proof sketch for Theorem 3. We sketch the proof of Theorem 3.

Exfiltration resistance. The bias-resistance of the key-generation protocol implies that the value $R \in \mathbb{G}$ that $\mathcal{F}$ holds at the end of Step 1 of the signing protocol is distributed statistically close to uniform over $\mathbb{G}$. If $R = g^r$, then the value $R_{\text{abs}} = g^{\pm r}$ is distributed exactly in the real ECDSA signing algorithm.

Having fixed R_{abs}, there are only two possible valid signatures that $\mathcal{S}^*$ can output that are consistent with this choice of R_{abs}: σ and $\bar{\sigma}$. In our protocol, $\mathcal{F}$ outputs one of these two at random. (Since $\mathcal{S}^*$ never causes $\mathcal{F}$ to abort, $\mathcal{S}^*$ must have sent a valid signature.) The signatures output by $\mathsf{ECDSA.Sign}$ are distributed identically, since the ECDSA signing algorithm outputs σ or $\bar{\sigma}$, depending on the "sign" of r modulo q.

Zero knowledge. Given a firewall $\mathcal{F}^*$, the simulator $\mathsf{Sim}(\mathsf{pk}, m, \sigma)$ operates as follows:

- Use pk, m, and σ to solve for $R_{\text{abs}} = g^{\pm r} \in \mathbb{G}$ using Equation (1) of Appendix B.
- Choose $R \xleftarrow{R} \{R_{\text{abs}}, 1/R_{\text{abs}}\}$ at random.
- Use the simulator for the key-generation protocol (given in the proof of its zero-knowledge property) on input R, with $\mathcal{F}^*$ playing the role of the browser, to get a transcript τ of $\mathcal{F}^*$'s view in the key-generation protocol. If the simulated protocol aborts output τ. Otherwise, output (τ, σ). To invoke the simulator, we use the fact that $\mathcal{F}^*$ never causes the honest signer to abort.

To show that the simulation is correct, we argue that (a) the simulation of the key-generation protocol is correct, using an argument similar to the ZK argument for the key-generation protocol and (b) if the key-generation protocol is bias-free then R is distributed uniformly over $\mathbb{G}$ (as in the simulation). Thus, the entire simulation is correct. □