A Cryptographic Dead Person's Switch

Owen McCadden and Houlton McGuinn

May 2022

Our project was the creation of a cryptographic dead person's switch, which automatically releases its contents should a user stop checking in with the service. Potential users of our project include whistle-blowers, political dissidents, and end-of-life planning for individuals. Our project makes use of verifiable delay functions (VDFs) and threshold secret sharing to achieve our security goals. Through the security properties of our VDF, threshold secret sharing, and modeling SHA256 as a random oracle, as long as some threshold of servers are honest and available the contents cannot be released prior to the users specified timeframe and are assured to be released.

1 Introduction

A dead person's switch is a system that releases its contents after a fixed amount of time only if the user stops checking in with the system. Each time the user checks in, the internal clock is reset. If the internal clock expires without a valid check in, the dead person's switch is triggered and the contents are released.

Simplified implementations of a dead person's switch rely on a trusted third party, leading to issues of trust and privacy. For instance, the third party could read the message before the internal clock expires, or the third party could discard or modify the message. Our implementation aims to address these issues and improve the dead person's switch using cryptographic techniques.

1.1 Background

Our implementation relies on verifiable delay functions and threshold secret sharing to achieve our security goals. Secret sharing involves splitting some secret S, into n shares such that k shares are needed to reconstruct S. We use a Python implementation Shamir secret sharing, the details of which can be found in [1] and [2].

Verifiable delay functions are a class of functions that require a specified number of sequential steps to evaluate, produce a unique output, and can be efficiently verified [3]. Importantly, they cannot be significantly parrelized or sped-up. Our projects uses a Rust implementaion of the Wesolowski VDF which is based on repeated squaring over a finite group of unknown order[4][5].

2 Overview

2.1 Motivation

The motivation for our project is to remove the need for a trusted third-party in a dead person's switch. Online services like Dead Man's Switch [6] and Dead Man Tracker [7] exist, but do not secure one's contents from the service. Given they release the plaintext without interaction from the user, they also have access to it. There is no guarantee that the contents released are the same contents submitted by the user. These services also provide a centralized failure point which is undesirable in the case of technical issues for the service or targeted failure should the service providers choose not to release a user's information. Our project aims to solve both these problems, hiding the plaintext from the service and ensuring the release of the information.

Real world use-cases of our project include end-of-life planning, for example sending loved ones the passwords and information required for accessing the deceased accounts. Another use case is as a form of insurance for political dissidents who may fear imprisonment or bodily harm. In 2010, WikiLeaks published an encrypted file labeled "insurance" to be released if something were to happen to their website or Julian Assange [8]. The encryption key for this file was escrowed with trusted WikiLeaks volunteers, our project removes the need for this trust in an individual.

2.2 Threat Model

The threat model for our project is focused on limited subsets of dishonest servers and clients who seek to learn the contents earlier than intended. Under our threat model, a single malicious server that deviates from the protocol should learn nothing about a user's contents earlier than intended and should not be able to prevent the release of or modify the contents. A semi-honest client should not be able to convince the servers to release the contents earlier than intended and should not be able to convince the servers that they have solved the VDF with a proof without having solved the VDF.

2.3 Security Goals

Our project has two primary security improvements over a standard dead person's switch: the service should learn the plaintext no earlier than when it should be released and the user should have some assurance that the plaintext will eventually be made public.

2.4 Security Assumptions

We assume that some fraction of the servers are honest and available. We also assume that SHA256 acts as a random oracle.

3 Protocol

While not implemented in our inital version of the protocol, all communications between the server and user should be encrypted.

Setup(user, share, public_key, pp): To initiate a dead person's switch, a user starts by generating 16 random bytes to be used as the VDF's public parameters. An encryption key is then derived using PBKDF2 and the

user's contents are AES256 encrypted using this key. The client also generates an elliptic curve public/private keypair. Finally, the ciphertext is padded and split using Shamir secret sharing into n shares of which k are needed to reconstruct the ciphertext. The client submits to each server: an unique identifier, a share of the ciphertext, it's public key, and the VDF's public parameters.

Check in(user, signature, new_pp, new_proof, new_share): A user prepares to check in by generating 16 new, random bytes to be used as the VDF's public parameters. A new encryption key is derived and the plaintext reencrypted. The new ciphertext is split into shares and signature using the user's private key. The user submits it's identifier, signature, new share, and new public parameters. A server accepts the check in if the signature is valid and updates its information for the user.

Request(user) $\rightarrow$ pp: Anyone can request the public parameters for a user. The server responds with the public parameters for the user's VDF function.

Solve(user, proof): $\rightarrow$ share $\vee\perp$ Anyone can attempt to present a proof that they have solved a user's VDF function and know the user's encryption key. Because of the VDF's properties, this is also proof that the time specified by the user has elapsed. If the proof is correct, the server returns its share, otherwise it returns $\perp$. Upon receiving the shares, the ciphertext can be reconstructed and the plaintext decrypted.

3.1 VDF & Key Generation

The user's setup in more detail:

1. Generate 16 random bytes.
2. Compute output of VDF using the 16 random bytes as the challenge.
3. Derive encryption key using PBKDF2 with the 16 random bytes as salt and the VDF output as key material.
4. Encrypt plaintext using AES-GCM with fixed nonce (each key is only used once).
5. Pad the ciphertext and tag to multiple of 128 bits.
6. Split the padded ciphertext of length l into $l/16$ shares (the Shamir secret sharing implentation used can only split equal to 16 bytes).
7. Secret share the split, padded ciphertext into n shares of which k are needed to reconstruct it.

Solving a user's switch in more detail:

1. Request the VDF's starting public parameters from the server for a user.
2. Begin solving the VDF. The client should also periodically re-request the VDF's public parameters as if they have been updated, the server will reject the proof of work on the old VDF.
3. Use PBKDF2 with the VDF output as key material and the VDFs challenge as the salt to derive the encryption key.

4. Present the VDF proof to the server which verifies it.

5. Combine the shares from each server to obtain the padded ciphertext.

6. Unpad the ciphertext and tag.

7. Decrypt the ciphertext using the derived key, obtaining the plaintext.

4 Discussion

4.1 VDF Choice & Implementation

For use in our project, we used an implementation of the Wesolowski VDF in Rust [4][9]. For this VDF, a user generates a public and private keypair. The user can then compute the trapdoor function using its secret key to efficiently compute the output y, given some data and a time parameter Δ. Anyone, given the public key, the data, and Δ can compute y only by expending time Δ. This allows a user to efficiently compute the output of the VDF, y, and therefore derive the encryption key and proof. To others, without the knowledge of the secret key, they will have to expend time Δ. This ensures that at least time Δ must occur before the encryption key computed and the shares retrieved from the server.

4.2 Preventing Early Disclosure

No one is able to learn the plaintext without first solving the VDF as the encryption key is derived from the output of the VDF. Because the VDF output is unpredictable and modeling SHA256 as a random oracle, the VDF must be solved to present the proof and learn the encryption key to decrypt the ciphertext. The properties of the VDF ensure that a certain amount of time has passed before anyone can learn the encryption key. A malicious server cannot cheat and learn the plaintext as it only has its single share and regardless would also have to solve the VDF to learn the encryption key. If multiple servers are malicious, the ciphertext can only be reconstructed if k of the n servers collude. Even if all servers are malicious, they still have to solve the VDF before learning the plaintext.

4.3 Assuring Disclosure

No action is needed by a user to make the encryption key public. Because the VDF parameters are public, should a user stop checking in, anyone can request and then solve the VDF challenge. Should a server be unavailable or otherwise unwilling to provide its share, the ciphertext can still be reconstructed if k of the n servers are available.

4.4 Key Derivation

For our project, we use the Wesolowski VDF that satisfies correctness, soundness, and sequentiality. While the output of a VDF is not indistinguishable from random, it is unpredictable by the sequentiality definition. Under the random oracle model, any unpredictable bitstring can be expanded to a longer, unpredictable uniformly random bitstring [3]. Assuming that SHA256 models a random oracle, and given that the key derivation material

is unpredictable because of the sequentiality of the VDF, no one can compute the encryption key without first solving the VDF.

4.5 Potential Issues

One potential issue with the protocol is the malleability of the shares. If k or more servers collude and replace the user's contents with contents generated by them and encrypted under the same key, there will be no way to tell. To prevent this, user's should digitally sign their ciphertext and publish this signature out-of-band somewhere.

For the information contained in a user's ciphertext to become public, someone has to compute the VDF and perform a long sequential computation. If no one performs this computation– potentially due to its length or lack of interest– the users plaintext will never become public. The role of VDF computer could potentially be done by the servers possessing the shares. Each server would immediately start solving the VDF function upon check-in and publish the plaintext to some website upon solving the VDF. This could potentially introduce performance issues with many users and may not be a feasible solution.

Our project in its current version allows anyone to setup a dead person's switch. There is also no way to know the value of solving the VDF without expending the computational resources required to solve it. An adversary could repeatedly setup new switches with bogus data as the output. As solving most VDFs would now be a waste, users may be disincentivized from spending computational resources to solve them. This could potentially be solved by requiring some cost to setup a switch (require a cryptocurrency transaction to occur first), require a user to prove their identity (but loses anonymity), or have a user vouch for their switch out-of-band (publish on twitter that switch 12345 is theirs and contains information of value).

Another potential issue is the overhead required to verify a submitted proof. While significantly cheaper than evaluating the VDF, a malicious client could launch a denial-of-service attack by submitting many incorrect proofs. A production version of this protocol would need to protect against this form of abuse.

Our project assumes that some fraction of the servers are available. As each user checks-in to each server individually, if a server is offline or otherwise unreachable, the state of the current share and VDF public parameters could become unsynchronized. While this is partially mitigated through the secret sharing and some fraction of servers being unavailable is okay, there is no clear way to recover from a bad state without a new check-in. Additionally, a client requesting the public parameters for a user could request from the server with the old parameters. While this is mainly a usability issue, ideally there would be some way to exclude a server with invalid state or at least notify it and the user that is is out-of-sync.

4.6 Improving Security With Hardware Enclaves

To improve the security of our implementation, a hardware enclave could

be used by the servers. This would provide additional security should a server become compromised, but importantly would enable attestation that the server's code matches a publicly known version. This, would decrease the likelihood that any given server is malicious or deviates from the protocol.

We explored the possibility of provisioning a hardware enclave using AWS Nitro Enclaves and AWS EC2. An AWS Nitro Enclave provides a secure, isolated compute environment within a parent EC2 instance [10]. The AWS Nitro hypervisor offers protection for the CPU and memory of the enclave. For our use case, we would also need to establish a secure local communication channel between the Nitro Enclave, the parent EC2 instance, and an external URL to allow an external user to communicate with the application. This fits well with a servers architecture in our protocol.

5 Conclusion and Future Work

5.1 Future Work

Our initial implementation of this project focused on the security goals for a user of the service. More work is needed on preventing abuse, particularly on denial of service attacks against a given server. Additionally, running the server within a hardware enclave will provide an added layer of security.

5.2 Conclusion

Using VDFs and Shamir secret sharing, our project creates an improved cryptographic dead person's switch, preventing the contents from being read earlier than intended and making the user more confident in their release. This project illustrates how cryptographic techniques can be used to improve the security and privacy of existing systems. We hope that our work and research can ultimately lead to a more secure, usable dead person's switch that does not rely on a trusted third party.

6 References

[1] Adi Shamir. 1979. How to share a secret. .ACM 22, 11 (Nov. 1979), 612–613. DOI:https://doi.org/10.1145/359168.359176

[2] Secret sharing schemes. Retrieved April 26, 2022 from https://pycryptodome.readthedocs.io/en/latest/src/protocol/ss.html

[3] Boneh, D. Verifiable Delay Functions. Crypto, 757-788 (2018). https://eprint.iacr.org/2018/601.pdf

[4] Wesolowski, B. Efficient Verifiable Delay Functions. J Crypto 33, 2113–2147 (2020). https://doi.org/10.1007/s00145-020-09364-x

[5] Boneh, D. A Survey of Two Verifiable Delay Functions. Cryptology ePrint Archive: Report 2018/712 https://crypto.stanford.edu/ dabo/pubs/papers/VDFsurvey.pdf

[6] Dead Man's Switch. Retrieved April 26, 2022 from https://www.deadmansswitch.net/

[7] Dead Man Tracker. Retrieved April 26, 2022 from https://www.deadmantracker.com/

[8] Kim Zetter. 2010. WikiLeaks posts mysterious 'insurance' file. (July 2010). Retrieved April 26, 2022 from

https://www.wired.com/2010/07/wikileaks-insurance-file/

[9] Poanetwork. Poanetwork/VDF: An implementation of verifiable delay functions in rust. Retrieved April 26, 2022 from https://github.com/poanetwork/vdf

[10] AWS Nitro Enclaves User Guide. Retrieved April 26, 2022 from https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html