DAWK: A Robustness Improvement to Prio

David Wu
MIT
wudavidh@mit.edu

Kevin Chen
MIT
kschen@mit.edu

William Xu
MIT
williamx@mit.edu

Aneesh Gupta
MIT
aneeshg@mit.edu

Abstract

In an age where information privacy is becoming ever more important, many data collection systems that preserve individual privacy have been proposed. Prio [1] is one such system that attempts to tackle this problem directly, and has been implemented in many settings, from Mozilla Firefox to smartphone COVID statistics aggregation.

However, the original Prio system relies on some relatively strict assumptions that must hold for the system to function properly. Through implementation and testing, these assumptions have been empirically been shown to be safe and viable in practice. Regardless, our group is interested in extending Prio to build in features that enable increased robustness against faults that could occur the system. The core of our approach comes from incorporating Shamir's Secret Sharing into Prio while preserving its core features.

1. Background

In this section, we present a summary of Prio, its key components, contributions, features, and constraints.

1.1. Prio

Prio [1] is a decentralized system for privacy-preserving data aggregation. Given a small number of servers and large number of clients, each of which with some secret value x_i, as long as a single server is honest, the Prio servers are able to compute $f(x_1, \cdots, x_n)$ without leaking almost anything about clients' private data, other than what the computed aggregate reveals. Unlike previous systems in this field, Prio is able to maintain correctness, robustness against malicious clients, privacy, and efficiency all at the same time, provided that its conditions are met. Specifically, Prio guarantees that if all servers are honest, the servers can compute $f(x_1, \cdots, x_n)$ with exact correctness, and, even with malicious clients, the scheme is robust enough to still provide correctness of $f(x_1, \cdots, x_n)$ with high probability. Additionally, if at least one server is honest, the values of x_i remain private and hidden from any adversary. These properties are especially useful when considering Prio's **efficiency**, as Prio is able to perform these computations privately while imposing minimal amounts of slowdown. Hence, when we modify the Prio scheme, we want to preserve as many of these properties as possible while relaxing the constraints on honest servers.

1.1.1 Prio secret-sharing scheme

Prio maintains client privacy by client data into randomized shares that are sent to servers: more specifically, Prio uses *additive secret-sharing*. Prio assumes all operations are performed in a field $\mathbb{F}_p$ of size p, where p is a prime. Then to split a value x into s shares, a client picks random integers $[x]_1, \cdots, [x]_s$ such that $x = \Sigma_i [x]_i$, and sends each share to one of the s servers.

The linear nature of additive secret-sharing means that servers can perform affine operations on shares without communicating: for shares $[x]_i, [y]_i$ and constants α, β, the servers are able to compute $[x]_i + [y]_i = [x + y]_i$ and $a[x]_i + b = [\alpha x + \beta]_i$ locally. Prio relies on this property to aggregate data: each server maintains an additive accumulator value that sums the validated shares it receives. Each server then publishes its accumulator value. Due to the linearity of the secret-sharing scheme, the sum of the accumulator values is equivalent to the sum of the clients' secrets. Thus the servers are able to compute this aggregate sum while maintaining client privacy.

Prio can also compute more complex statistical operations without major change to the accumulator process by manipulating inputs: instead of a client simply sending shares of its secret, it instead computes an encoding of its secret, splits the encoded value into shares, and sends those to servers. This encoding is chosen such that computing the sum of the encoded shares is equivalent to computing the desired statistic.

Using additive secret-sharing, we see that any subset of up to $s - 1$ shares reveals nothing about the secret x. This fact demonstrates the tension between *privacy* and *fault-tolerance* in Prio's secret-sharing scheme: the information-theoretic guarantee that no information about x can be recovered from any $s - 1$ or less shares implies that the result

of affine operations on x cannot be recovered from the result of equivalent operations on its $s-1$ shares. This means that all s servers must be honest and functional in order to aggregate data.

Prio circumvents this problem by relying on the assumption that servers are always honest and functional, reasoning that real-world deployments would only need a small number of servers existing in fixed locations with known administrators, and thus server robustness and trust issues could be handled outside of Prio itself. Our system strengthens server fault tolerance by replacing additive secret-sharing with a more robust scheme, as discussed in Section 3.

1.1.2 Prio verification scheme motivation

While Prio assumes no adversarial servers, a significant portion of its design involves defending against adversarial clients. Prio defines adversarial clients as those who submit inputs that corrupt the overall system output. For example, one natural area where this issue arises is voting applications. Say that users can cast at most 1 ballot for each candidate, meaning that each user should only be able to increase a candidate's vote count by at most 1. With no verification, a malicious user could send an input that increases a candidate's vote count by 100, overwhelming the data collected from other users and thus corrupting the output.

Prio deals with this issue through its verification scheme. As with many other privacy applications, Prio's increased privacy comes the cost of verification complexity: by nature of secret-sharing, servers should not be able to glean any information about client secrets based on the shares they receive, so it follows that they should not have sufficient information to check input validity on their own. This setting – where we must verify the truth of a statement without learning any other information about the statement itself – naturally lends itself to zero-knowledge proofs. Thus, Prio introduces a new type of zero-knowledge proof: *secret-shared non-interactive proofs* (SNIPs).

1.1.3 Prio verification scheme

Secret-shared non-interactive proof (SNIP) protocols involve interactions between a single prover and multiple verifiers (client and multiple servers). The servers hold shares $[x]_i$, while the client holds the secret $x = \Sigma_i[x]_i$. All parties hold a predicate: an arithmetic circuit Valid. The client must convince the servers that $\mathsf{Valid}(x) = 1$ without leaking information about x by sending a "proof string" to each server. The servers communicate upon receiving the proof string, then determine whether $\mathsf{Valid}(x) = 1$ or not. In this section, we present a SNIP constructed for use in Prio. For a more generalized SNIP construction, refer to Corrigan-Gibbs and Boneh [1].

The original Prio scheme verifies that user inputs are of the form $\{0,1\}^*$. Let v_i refer to the ith value of this n dimensional user input vector. Prio then defines two n-degree polynomial functions f and g. The functions have properties as follows for all integer i between 1 and n:

$$f(i) = v_i \tag{1}$$
$$g(i) = v_i - 1 \tag{2}$$
$$h(i) = f(i)g(i) \tag{3}$$

Since all $v_i \in (0,1)$, this means $h(i) = 0$. These polynomial functions form the basis for the verification scheme.

After a user generates their user vector v, they can easily compute f, g, and h. They will then send their shares to their respective servers, as well as the polynomial h.

Upon receiving h, both servers can check to make sure that indeed $h(i) = 0$ for all relevant values of i. If this is not the case, they reject the input. Next, the servers want to check that $f(x)g(x) = h(x)$, and one indirect way of probabilistically doing that is by checking that equality for some random value r. If $f(r)g(r) = h(r)$ for some truly random r, then there is a good chance that $f(x)g(x) = h(x)$ as long as the finite field is large enough.

To check this, assume we have generated some r that all the servers have agreed on. Given their shares, each server can perform LaGrange interpolation on their shares alone. They will each compute their own $f'(x), g'(x)$ with the properties specified above, but replacing v_i with their share of v_i. Once they have these polynomials, they can just plug in r to obtain their share of $f(r), g(r)$ which, when added to the share of $f(r), g(r)$ from the other servers, yields the true value of $f(r), g(r)$. They can now compute $\hat{h}(r) = f(r) \cdot g(r)$ using a variant on the Beaver multi-party computation protocol and compare it to $h(r)$, the value obtained from directly plugging in r to the provided h function.

If $\hat{h}(r) = h(r)$, this means that the client likely did not cheat and computed the h polynomial truthfully. The servers thus accept the user input and add it to their own accumulators. One thing to note is that since the polynomials are evaluated at some random r, this does not leak any information regarding the client secret.

1.2. Shamir's Secret Sharing

Shamir's Secret Sharing is a secret-sharing scheme that is based on polynomial interpolation over finite fields. At its core, this process relies on the fundamental property that k points is enough to uniquely define a polynomial of degree up to $k-1$. Given some secret value y_0, this scheme allows users to create n secret shares which require only k shares to reconstruct the secret value.

Assuming this fundamental property of polynomials, we can reconstruct the equation for a polynomial given k points

(x_i, y_i) by requiring the equation for the polynomial be exactly y_i whenever $x = x_i$. To do this consider the following summation:

$$f(x) = \sum_{i=1}^{k} y_i \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

By convention, the secret y_0 is usually set to $y_0 = f(0)$. From the above equation, we observe that the function f has the property that $f(x_i) = y_i$ because all the terms with the other y_i will be 0 in the summation. Additionally, the fractional term on the left will be equal to 1 for the y_i term of the summation. To obtain the secret value, users just need to recover the equation of the polynomial and plug in 0.

To generate shares for the setup where we have n users where any k can reconstruct the secret, generate a degree $k-1$ polynomial where $f(0)$ is the secret value and find n points (x_i, y_i) along that polynomial ($f(0)$ cannot be a share since that is the secret value).

2. DAWK Scheme

When we first analyzed the Prio system, we noticed that it relied on a few key assumptions. One very important detail is if a single server goes down completely, the information stored in the other servers is useless, and all data is irrecoverable (Section 1.1.1). All servers need to be functional in order for Prio to work properly. The problem of a single server going down and rendering Prio nonfunctional seemed like a significant problem, one that we hope to be able to address through our extension.

The key insight that we uncovered is that the original Prio system breaks up information into non-redundant shares. If a single share is lost, the secret value cannot be recovered. However, we know that Shamir's Secret Sharing is a secret-sharing scheme that builds redundancy into its shares. The idea is to use this to construct our shares instead of constructing the additive shares proposed in the original Prio system.

However, one immediate problem that we realized is that Prio actually relies heavily on the fact that the shares add up to the secret value in the finite field. This property of the shares enables Prio to perform its verification scheme, which ensures that shares are well-formed. Therefore, a natural idea is to try and convert the shares produced by Shamir's Secret Sharing to shares with this key property, allowing us to still utilize the properties of the original system.

To do so, consider the following. Let y_0 be the client secret, and f be a degree-$k-1$ polynomial where $f(0) = y_0$. Assign server i the share produced by Shamir's Secret Sharing at $x = i$. This means that each server is always responsible for holding the share at a particular x value. Server i thus knows their own secret share, (x_i, y_i), as well as the x_i for all the other servers. Taking a look at the equation for calculating the secret value in Shamir's Secret Sharing, we immediately see that each server is capable of computing a single term in the summation. If each of the servers performs this computation on their shares, we can restore the property that the shares add up the secret value.

$$f(0) = \sum_{i=1}^{k} y_i \frac{\prod_{j \neq i} -x_j}{\prod_{j \neq i}(x_i - x_j)} \tag{4}$$

$$y_i' = y_i \frac{\prod_{j \neq i} -x_j}{\prod_{j \neq i}(x_i - x_j)} \tag{5}$$

$$f(0) = \sum_{i=1}^{k} y_i' \tag{6}$$

With this additive property restored, we have now successfully created a scheme that has built in redundancy for its shares while simultaneously preserving the additive property that Prio relies on. If a server goes down, the remaining servers can still function by pooling their shares together to simulate the Prio scheme, allowing them to reconstruct the data and compute meaningful statistics on it.

Moreover, given this redundancy in our shares, we can also perform some consistency checks in our modified Prio system. For example, consider the case where we have a single malicious server that is broadcasting incorrect values during the verification scheme. In order to address this, we can simply perform the verification scheme twice, each time with a different subset of k online servers. If there is a malicious server giving junk for verification, we will find that the sum of shares for computing $h(r)$ differs across the two subsets.

By detecting this discrepancy, we can check more k-subsets until a match with a previous subset is achieved. We can be certain that the matching sum is the correct sum with very high probability. Furthermore, we can then check which subsets result in incorrect sums to detect and remove malicious servers.

2.1. Verification

To ensure robustness, we use Shamir Secret Sharing on all SNIP inputs. Prio depends on each server getting a share of each of $(f(0), g(0), h, a, b, c)$ from the clients. Instead, we have each client send a Shamir Secret Sharing share with threshold $k-1$ to each server. This means each server has $(f(0)_i, g(0)_i, h_i, a_i, b_i, c_i)$, where any k servers can reconstruct the original values. Then prior to verification, a subset of k functioning servers are randomly chosen to perform the verification. These servers all compute the Lagrange Interpolation on their six values to get

$(f(0)'_i, g(0)'_i, h'_i, a'_i, b'_i, c'_i)$ such that

$$(f(0), g(0), h, a, b, c) = \sum_{i=1}^{k} (f(0)'_i, g(0)'_i, h'_i, a'_i, b'_i, c'_i)$$

, where the subscripted i represent the k chosen servers. These modified shares are then used in the Prio verification scheme to check whether the provided shares are valid. Note that multiplications between $f(r)_i$ and $r \cdot g(r)_i$ are computed using Beaver's multi-party computation protocol [2]. Then, other sets of servers with size k are checked until all online servers have been used in verification. This ensures that every server's secret share has been verified.

The servers are not checked for adversarial possibilities in the verification stage. The servers can follow one of four possibilities: honest, honest but curious, faulty, or malicious. If all servers are honest, then everything runs as intended. If there are honest but curious servers, validation occurs properly as long as at least one of the chosen k servers is honest. If a server is faulty, then it will not be selected to perform the verification. Unfortunately, malicious servers can impact whether a client's value is verified.

However, both malicious clients and malicious servers have their impact reduced due to randomness as long as they are not cooperating. A malicious client may attempt to send faulty shares of the verification inputs to some servers, but since all online servers are verified, the malicious client's share will be rejected. A malicious server could make a valid share not pass validation with 100% chance and any share not pass validation with $\frac{|\mathbb{F}|-1}{|\mathbb{F}|}$ probability if $\mathbb{F}$ is the field used to perform the SNIP. We could attempt to catch these malicious servers in the validation phase, but it is way more performance intensive than trying to catch them in the computation phase. This is due to the fact that computing SNIPS is more costly than aggregation. Thus we leave catching malicious servers to the computation phase.

We must also consider the adversarial case where both a malicious client and malicious server are working together. While in the case where we analyze maliciousness separately, it is essentially impossible for a malicious server to validate a bad share (since the server must determine $f(0)$, $g(0)$, and $h(0)$ without outside help), here it is much easier. Due to optimizations put in place for Prio (specifically the fixed r), DAWK cannot handle malicious client-server communication without a significant hit to performance.

In fact, with this fixed r, neither Prio or DAWK can handle malicious client and honest but curious server either. This is because the honest and curious server can send the fixed r to the client. Then, the client can choose $f(x)$ and $h(x)$ that have zeros at $x = r$ and the algorithm will return that the client's share is valid, no matter the value. Thus, the DAWK scheme cannot handle malicious clients working with honest and curious servers.

A server can also fail during the validation scheme. This can occur both naturally, or as a result of an adversarial targeted attack. If a server fails, verification cannot go forward as the interpolated shares are no longer valid since verification requires k working servers. Thus, a new random k sized subset of working servers is tried until a result is achieved.

In general, to avoid unverified secret shares being used in computation, we run the verification on all online servers. However, if a server goes down during the verification phase, then comes back online afterward, its unverified secret share could be used in the computation, which could contaminate the data. To avoid this scenario, we would only mark that server as back online once computation on the shares has completed.

3. Analysis

The three main properties discussed in the Prio paper were robustness, performance, and privacy. The system maintains high privacy and performance, but stated they did not wish to improve robustness since that would result in much worse privacy and performance. Here, we look at how much the robustness, performance, and privacy actually were impacted by the DAWK modification.

3.1. Robustness

Unlike Prio, where a single compromised or unavailable server would break the algorithm, DAWK is robust against multiple servers going down. Specifically, if DAWK is being run on n servers, s minimum of $n - k + 1$ servers need to fail in order for DAWK to be unable to function. This is much better robustness than found in Prio.

Furthermore, DAWK is able to detect and remove malicious servers. By verifying aggregation results, DAWK can find malicious servers and remove them from the list of viable servers. Keeping this work in the aggregation stage allows the detection process to occur quicker and more reliably than in the validation stages.

3.2. Performance

In our modified scheme, servers need to perform a few computations in order to convert the new Shamir's secret shares to the Prio shares with the additive property. To do these computations, the servers just need to multiply $O(k)$ values which does not add significant overhead compared to the previous Prio system.

After performing these transformations on the new shares, we convert all the shares to the original Prio system, meaning that all the work going forward is exactly the same as the Prio system. Critically, this means that the amount of communication cost incurred between servers is still just a constant amount. Since performing Lagrange interpolation

already requires $O(k)$ work, this additional work per server is not significant in the overall scheme.

If we include the modification that enables us to perform consistency checking for malicious servers, this increases the cost of communication since the verification procedure will need to be run at least twice. If there are no malicious servers, we can stop here and incur no more verification costs.

3.3. Privacy

Since we are able to convert the shares to shares with the additive property, we are able to ensure that we can preserve all the original privacy guarantees of Prio. Indeed, an adversarial actor still needs to access k servers to obtain information about the individual results. The difference with our approach is that since there are more than k servers, there are more possible servers susceptible to being controlled by an adversary. This could be a real downside if servers went malicious randomly. However unlike servers failing, it is unlikely that a server turns malicious without an outside actor. Thus, an outside actor would need to gain control over k servers, meaning that the work needed to be done by an adversary is the same as in Prio.

Thus, we believe the robustness against a single random failure of a server is more important than privacy in a targeted attack on k or more servers. Since so many servers need to be attacked and malicious servers can be reported in the aggregation stage, we believe such an attack would be detected in time to preserve privacy. Furthermore, if we wanted to be even more secure against malicious servers, we could implement other linear secret-sharing multi-party communication methods like MP-SPDZ [2]. However, we believe our approach to detecting malicious servers is sufficient.

4. Conclusion

In this paper, our group was able to produce a robustness extension to the Prio system. By leveraging redundancy in the Shamir's Secret Sharing scheme, we are able to improve the robustness of the original Prio scheme. There are many benefits to having this extension, such as the ability to detect malicious servers, and the ability to continue functioning in the presence of server outages.

However, we also note that DAWK makes some tradeoffs when providing additional robustness guarantees. Most notably, these tradeoffs come in the form of increased communication and computational costs. The servers are required to perform more computations and communicate more in this scheme compared to the original Prio system, which was able to guarantee privacy without incurring much slowdown.

References

[1] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, Boston, MA, Mar. 2017. USENIX Association. 1, 2

[2] Marcel Keller. *MP-SPDZ: A Versatile Framework for Multi-Party Computation*, page 1575–1590. Association for Computing Machinery, New York, NY, USA, 2020. 4, 5