Formal Model and Verification of Privacy-preserving String Discovery for Mobile and Web Analytics

Istemi Ekin Akkus

iakkus@mpi-sws.org
Max Planck Institute for Software Systems

Technical Report MPI-SWS-2014-006
September 2014

1 Introduction

Web and mobile app analytics fuel an important part of the Internet economy and are important for many types of analysts, including web publishers, mobile application developers, advertisers and marketing companies. While statistics about user demographics (e.g., age, gender, income) are important, a new class of statistics is emerging: arbitrary text values or *strings*. Imagine a website publisher who wants to learn which (previously unknown) search phrases (e.g., 'pizza nearby') are used by how many of its visitors, or consider the developer of a photo application that may want to learn about the free-text tag values its users assign to their photos (e.g., 'dad and the cats'). Other examples include URLs visited, names of products viewed and names of installed applications.

The straightforward approach to tackle this problem is to use a third-party data aggregator; however, it requires tracking of users. While beneficial for analysts and aggregators, this approach comes with a price for user privacy: Aggregators can build profiles about individual users [30] with the sensitive information they obtain. Users' trust in aggregators not to misuse this information has been violated in many cases [5, 6, 8]. In response, researchers and industry have proposed methods to detect and prevent tracking [1, 3, 9–11, 30]. While these methods protect privacy, they significantly reduce the benefits of analytics data and limit the information analysts learn about users.

Some systems [15, 16] try to tackle these privacy concerns via general-purpose secure multiparty computation (SMC) protocols [16], or expensive cryptographic operations [15], such as oblivious transfers (OT) [29]. These operations put a significant load on the clients, whose resources may be limited in large-scale, distributed environments such as the web. In fact, users increasingly access the web via mobile devices with limited capabilities compared to personal computers [2, 4]. An inability to support such clients will hinder the use of these systems for privacy-preserving analytics on the web. Additionally, the original goal of these systems is to aggregate and correlate network events among big organizations (e.g., ASes). This specialization limits the length of strings these systems can handle due to the underlying cryptographic primitives. For instance, Sepia [16] and Applebaum et al. [15] assume a string length of 32 bits (i.e., IPv4 address). To support longer strings as in our examples, these systems would require substantial changes.

Recent proposals for privacy-preserving analytics on the web avoid the inherent trade-off between privacy and scalability: With the help of a client software, they store user data at users' devices and release it in a protected fashion, either with proxies [14, 17, 18, 24] or restricted interfaces [26]. These systems utilize less sophisticated, but faster crypto operations than SMC or OT, scaling to the web. However, they require the analyst to enumerate a set of string values that are relevant to the user data in question, either as potential answers for analysts' queries over user data [14, 17, 18] or as counter names [24, 26]. This requirement limits these systems' applicability: in many analytics scenarios, like our search phrases and photo tags examples, pre-defining an exhaustive list of such values in advance may be difficult or impossible.

Recently, we proposed a system that enables analysts *discover* previously unknown string values of arbitrary length while supporting even the most resource-constrained user devices [13]. The user data resides at each user's own device running a client software. The client periodically participates in string discovery procedures by submitting its encrypted strings. These strings, while still encrypted, are then counted by the entity providing the discovery service (i.e., the aggregator) and two honest-but-curious proxies. Strings with a (noisy) count above a threshold are then decrypted and provided to the analysts.

In this report, we present the formal model of this system [13], and verify its protocol using ProVerif [7]. For our formal model, we place the adversary separately at each component and describe the variables that are of interest to the adversary. We let ProVerif overapproximate the state space of the protocol to discover any attacks, and reason about the potential attacks it finds. We show that these attacks are false. We also describe any limitations of ProVerif regarding the attacks it can and cannot find.

The next section presents an overview of our system's operation along with the definitions, assumptions and components of our system. Section 3 describes our formal tools. Section 4 explains the primitives we use in our model. We present and explain our model in Section 5. We discuss our model's limitations in Section 6, and conclude in Section 7. All model files along with documentation can be found at the following address: `https://www.mpi-sws.org/~iakkus/private/verif/`

2 System Overview

2.1 Definitions

We define the following terms: A **string** is a text value present at the user's device. Some examples are 'google.com', 'pizza nearby' and 'enjoying spring in Paris'. A **string type** is the class of the string. The string instance 'google.com' may be of string type 'visited websites'. Similarly, 'pizza nearby' may be of type 'search phrases' while 'enjoying spring in Paris' may be a 'photo tag'.

2.2 Privacy Definitions

Our privacy notion is based on Pfitzmann and Köhntopp [28]: The participation of the clients to a string discovery procedure should be anonymous, such that given a string value or type, no component should be able to associate it with a client. It should also be unlinkable, such that given two string values or types, no component should be able to tell if they are from the same client. In addition, a discovered string should not reveal any information about any other string. For example, guessing a common string value should not leak any information about any other string.

Furthermore, our system aims to ensure that a string is discovered, only when there is a sufficient number of clients with that string. Given this restriction, an adversary can artificially inflate a string's count with fake clients. Unfortunately, it is not straightforward (if not impossible) to model such counts using ProVerif. We model this notion of privacy with the adversary's final goal in mind: to learn the existence of a real client with a rare string.

2.3 System Components

There are three types of components in our system: client, aggregator, and proxies. Clients and the aggregator already exist in today's aggregation infrastructure. Proxies have been widely proposed for privacy purposes [15, 17, 18, 23, 24], and we also adopt this approach.

Client. The client is a piece of software that stores user data (i.e., strings) locally, like other systems [14, 17, 18]. The client also keeps metadata regarding string types, and participates in discovery procedures by sending its encrypted strings.

Aggregator. The aggregator provides the string discovery service. This service reports previously unknown strings and their noisy counts to the analysts, who may express their interest in learning strings of a string type (e.g., to query distributed user data with other systems, such as [14, 17, 18]). The aggregator handles all interactions with the analysts, and controls access to the discovered strings (e.g., shares strings of an analyst-specific type only with that analyst).

Proxies. The proxies provide clients with network anonymity, and enable the aggregation of encrypted user data and discovery of strings. They also help the aggregator limit malicious clients' effect on string counts.

2.4 Trust Assumptions

We assume that the aggregator and the proxies are honest-but-curious: they follow our protocol, and do not collude with each other. However, each component may operate fake clients to try to link or deanonymize other client strings.

Although our honest-but-curious model is weaker than a more general model (i.e., arbitrarily malicious aggregator and proxies), we think that it reflects the reality on the Internet: The aggregator operates a business by providing string discovery service for analysts. The proxies can be operated by independent companies and/or privacy watchdogs. All of these entities would put their non-collusion statement in their privacy policies, making them legally liable. Moreover, any entity not following the protocol would risk losing reputation and customers. Previous systems make similar assumptions [14, 15, 17–19, 23, 24].

Finally, we assume the aggregator and proxies are not impersonated, and all end-to-end connections use TLS (i.e., no eavesdropping and no in-flight modifications).

The client typically runs on a user device, but may also run on a different, trusted platform. Like previous systems [14, 17, 18], we assume that the user trusts the client to protect the data it stores and operates, just as users trust their browsers for certificate handling and TLS connections.[1] By contrast, a client can act maliciously towards the aggregator and analysts, and try skewing string counts by sending the same string multiple times.

[1] We do not protect against malware infections on user devices, which can violate user privacy in many ways.

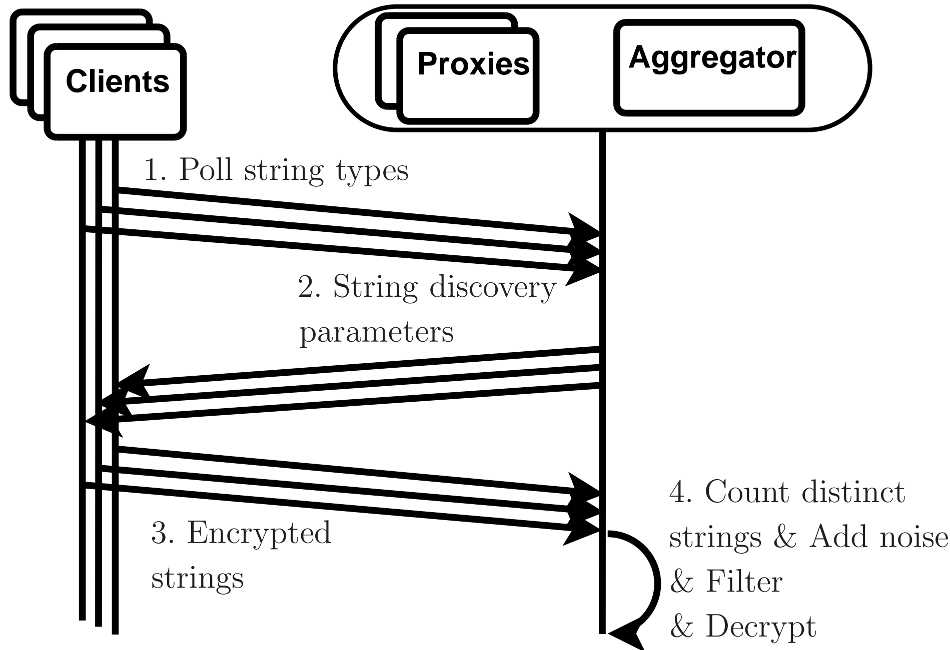

Figure 1: Overview of our system's operation.

2.5 Operation

Figure 1 shows an overview of our system. The aggregator periodically runs string discovery procedures. Clients periodically poll the aggregator with their string types (step 1 in Figure 1). These polls are XOR-encrypted and sent via the proxies to provide clients with anonymity and unlinkability: the aggregator cannot associate clients with string types, and cannot tell if any two requests are from the same client. The proxies also cannot learn a client's string types because of the encryption.

After receiving a poll request for a string type, the aggregator sends the associated (XOR-encrypted) string discovery parameters to the client via the proxies (step 2). The parameters include the ϵ value (used for Laplace noise) and the discovery epoch. The discovery epoch is used to synchronize the start and end times of discovery procedures for many string types. This synchronization enables the aggregator to group multiple string types together, such that an adversary cannot deduce a client's string type from the participation in the discovery procedure. It also serves as a checkpoint for the duplicate detection to limit malicious clients.

After getting the parameters, the client retrieves the strings belonging to the string type from its local database. The client then XOR-encrypts each distinct string separately before sending it for aggregation (step 3).

During the aggregation step, our system utilizes a low-cost comparison method that only reveals if any two XOR-encrypted strings are equal. With this method, our system counts distinct strings, adds noise to their counts, and applies the discovery threshold—all without learning the actual string values. Strings whose noisy count pass the threshold are then decrypted (step 4), and the aggregator reports them with their noisy counts to the analysts.

Table 1: Process grammar.

$P, Q :=$	processes
0	null process
$P \vert Q$	parallel composition
$!P$	replication
new $n : t$; P	name restriction
if M then P else Q	conditional
let $x = M$ in P else Q	term evaluation
in(M,x:t); P	message input
out(M, N); P	message output
$R(M_1, ..., M_k)$	macro usage

A malicious client can try to exploit our system's anonymity and unlinkability properties to skew a string's count by sending it multiple times. Our system checks for such duplicates, potentially reported by any client, before counting strings. We utilize the same low-cost, blind comparison method mentioned above, and detect malicious clients without violating the privacy of honest clients.

3 Formal Tools

Before we present the primitives used in our system and their equivalents in our formal model, we describe the formal tools we use to model our system and prove its privacy properties.

3.1 Applied Pi Calculus

Pi-calculus [27] is a language that is used to formally model distributed systems and reason about their interactions. Applied Pi-Calculus [12] is an extension of pi-calculus that is used to model and reason about cryptographic protocols. These distributed systems are modeled as a collection of parallel processes that exchange messages using channels.

Here, we describe some basics of the language (in conjunction with ProVerif) and how it is used to model interactions among concurrently running components of a system. The details of the language and how it is used in ProVerif can be found in the ProVerif manual [7].

3.1.1 Processes

A process is used to model the logical actions of the components in the system. The grammar to build processes is given in Table 1.

3.1.2 Messages

Processes interact using messages. A message can be a name, a variable or the output of a constructor, or a combination (i.e., tuples). A name (e.g., 'string1') is used for atomic data. A variable (e.g., x) can be bound to a name or a message. Equivalence of two messages can be learned by applying an equation of the form $x = y$.

3.1.3 Constructors/Destructors

A constructor is a function that can be applied to names, variables and other messages. The corresponding destructor of a constructor ensures that a message can only be reversed into its original content (i.e., name, variable, message), if and only if the correct conditions are present.

For example, one can model a secure, one-way hash function as a constructor without a destructor. Because there is no destructor, the output of this constructor cannot be reversed.

```
type hash.
fun H(bitstring): hash.
```

On the other hand, we can model the symmetric encryption *senc* with a destructor *sdec*, such that it will only output m when the key used to decrypt (i.e., k) is the same key used in the constructor *senc*.

```
type key.
fun senc(bitstring, key): bitstring.
reduc forall m: bitstring, k: key; sdec(senc(m,k),k) = m.
```

Another way to model certain cryptographic primitives is to use equations. For example, one can model the symmetric encryption/decryption above as equations that capture the relationship between the constructors for all variables as:

```
type key.
fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
equation forall m: bitstring, k: key; sdec(senc(m, k), k) = m.
equation forall m: bitstring, k: key; senc(sdec(m, k), k) = m.
```

3.1.4 Channels

Messages can be output and input on channels. Channels are asynchronized, such that the messages sent on a channel can be received out of order. A message m can be sent on a channel c using $out(c, m)$. Similarly, it can be input from the channel $in(c, r)$, such that r will be bound to the message received from the channel. If the message m is a tuple of form (x, y), then the input action can be performed with a conditional, such that the variable y will be bound to variable z in $in(c, (= x_2, z))$ if and only if $x = x_2$.

3.2 ProVerif

ProVerif [7] is a tool for automated analysis of cryptographic protocols. Distributed systems modeled in applied pi-calculus can be automatically analyzed to prove secrecy properties of these systems. ProVerif has been widely used in the literature to analyze properties of various cryptographic protocols (see the ProVerif manual [7] for a complete list).

ProVerif can perform reachability analysis of properties on an unbounded number of instances of the protocol. To do so, ProVerif overapproximates the state space of the protocol and explores it. As a result, when ProVerif claims that a property is true (i.e., no attack is possible), then it is true. In other words, ProVerif is sound. If ProVerif can prove a property is false, it generates an attack

trace on why the property is not true. However, due to the overapproximation of the state space, it is possible that ProVerif thinks there is an attack, although there is no attack possibility. In these cases, the attack trace can be investigated to confirm whether the attack is real or false. If ProVerif cannot prove that the property is neither true nor false, ProVerif states so.

ProVerif provides use of private channels. These channels are especially useful for modeling end-to-end encrypted channels between components, where the adversary is assumed not to have access to the messages.

3.2.1 ProVerif Limitations

Although ProVerif is sound, it is not complete. That means, ProVerif may not be able prove that a property holds. It is also possible that ProVerif generates a false attack due to the overapproximation of the state space, in which cases manual investigation of the attack trace may be required to determine whether the trace is actually valid.

As stated above, ProVerif performs the analysis with an unbounded number of sessions of the protocol. However, the repetition of actions cannot be supported, because repeated actions are translated into the same Horn clause as non-repeated actions. As a result, there is no method for counting how many instances of the protocol ran until a point. This limitation prevents modeling of an adversary that might delay its attack until only after a certain number of messages have been received.

ProVerif cannot model traffic analysis, and privacy properties based on the 'hiding in the crowd' principle cannot be modeled. Although a piece of information may not be meaningful in probabilistic sense, the mere fact that the adversary has access to it will trigger ProVerif to generate an attack trace,

4 Formal Primitives

In this section, we describe the primitives we use throughout our model.

4.1 XOR-encryption

Our system uses XOR as its crypto primitive like SplitX [17]. Splitting is equivalent to encryption, and joining is equivalent to decryption. These operations enable a source to anonymously send a string to a destination via two different relays, without the relays learning the string due to the encryption. Meanwhile, the crypto operations for the source and destination are low-cost (Figure 2).

To send a string S to a destination, the source splits S to obtain two split messages, X and R. Let L be the length of S. The source first generates a random string R of length L with a Pseudo Random Number Generator (PRNG) and a secure *seed* (used only once), and encrypts S with R:

$$\begin{aligned} R &= PRNG(seed, L) \\ X &= S \oplus R \end{aligned}$$

The source also generates a *split identifier* (*sid*). *sid* is a large random number (e.g., 128 bits), and ensures the two split messages, X and R, will be uniquely paired by the destination with high

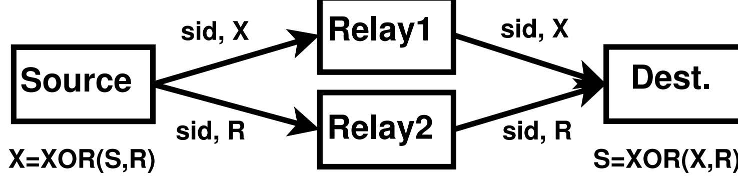

Figure 2: Splitting and joining. S is split to X and R. They are sent with the same sid via two relays.

```
(* XOR-encryption and decryption *)
(* same as symmetric encryption/decryption *)
fun split(bitstring, bitstring): bitstring.
fun join(bitstring, bitstring): bitstring.
equation forall s: bitstring, r: bitstring; join(split(s, r), r) = s.
equation forall s: bitstring, r: bitstring; split(join(s, r), r) = s.
```

Figure 3: Formal definition of splitting and joining in ProVerif.

probability. The source then sends X and R to the relays, who forward them to the destination:

$$\begin{aligned} Source \rightarrow Relay_1 \rightarrow Destination &\ : \ sid, X \\ Source \rightarrow Relay_2 \rightarrow Destination &\ : \ sid, R \end{aligned}$$

Borrowing notation from [17], we denote the split message pair $\{X, R\}$ as $\underline{S}$ (underlined S), and write:

$$Source \xrightarrow[Relay_2]{Relay_1} Destination : \underline{S}$$

The destination joins the split messages to obtain S:

$$S = X \oplus R$$

ProVerif cannot fully model XOR operation. Fortunately, our split and join operations can be modeled simply as symmetric encryption and decryption (Figure 3).

4.2 Pairwise-XOR Hash (PXH)

To count distinct string values without revealing them, our system uses a blind comparison method to determine the equality of any two XOR-encrypted strings. Consider two strings S_i and S_j with split message pairs $\{X_i, R_i\}$ and $\{X_j, R_j\}$, and split identifiers sid_i and sid_j, respectively. Recall that the split messages are held by two relays (i.e., X_i and X_j by Relay$_1$, and R_i and R_j by Relay$_2$). Let H be a secure hash function (e.g., SHA-2). For each relay, we define the pairwise-XOR hash (PXH) operation as:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &= H(X_i \oplus X_j) \\ PXH_{Relay_2}(sid_i, sid_j) &= H(R_i \oplus R_j) \end{aligned}$$

```
(* special type for the PXH result *)
type PXH.
(* comparison operations *)
(* because of secure hash, they are irreversible; thus, no destructors *)
(* PXH operations for aggregator and proxy1 *)
fun computePXH(bitstring, bitstring, bitstring): PXH.
(* with no secret (for the ‘‘known R values’’ attack by the adversary at proxy2 *)
fun computeKnownPXH (bitstring, bitstring): PXH.
```

Figure 4: Formal definition of the PXH operation. The third parameter is the secret shared between the collecting components (i.e., in our model, Proxy$_1$ and the aggregator).

Recall that $X_i = S_i \oplus R_i$ and $X_j = S_j \oplus R_j$. Therefore:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &= H((S_i \oplus R_i) \oplus (S_j \oplus R_j)) \\ PXH_{Relay_2}(sid_i, sid_j) &= H(R_i \oplus R_j) \end{aligned}$$

If $S_i = S_j$, then $PXH_{Relay_1} = PXH_{Relay_2} = H(R_i \oplus R_j)$. By comparing PXH_{Relay_1} and PXH_{Relay_2}, our system can *blindly* determine if the original strings S_i and S_j are equal. Compared to just using the pairwise-XOR value, the secure hash ensures that one string cannot be reverse-engineered, even when strings are unequal and the other string is easily guessed (e.g., a common string).

Our PXH operation is straightforward; however, modeling it using ProVerif is not: Equal strings are supposed to cancel each other out; however, due to lack of support for such functionality of XOR in ProVerif, this operation cannot be modeled this way. Although there has been work on how to reduce protocols that use XOR semantics to a non-XOR version, such that ProVerif can be utilized [25], we choose a much simpler approach to overcome this limitation.

We make use of two special channels, and ensure that these channels are private and thus, not accessible to the adversary (Figure 5). These channels essentially help us emulate the ideal functionality of the PXH operation, in which equal string cancel each other out. Our special channels work as follows (Figure 6): Clients output their string values and split identifiers (i.e., *sid*) to a channel (i.e., *csid_str_map*). The aggregator outputs the PXH value of two encrypted strings along with the $\{sid, sid\}$ tuple to another channel (i.e., *cpxh_sid_sid_map*)[2]. The comparing proxy (in our model, Proxy$_2$), uses the PXH value to retrieve the original *sid* values of the strings and then compares their values. This way, we can emulate lacking functionality for XOR and determine whether any two strings are equal.

Note that these channels are only used for the PXH comparison to determine the equality of encrypted strings. Even if we place the adversary at Proxy$_2$, our model ensures that the adversary does not have access to these channels neither to the variable values obtained from these channels.

4.3 Datastores

We model a component's datastores as private channels. These datastores enable us to store the state of a component. We encode state information using messages and use the channel as a key-value

[2]One could have also used the other collecting component (in our model, Proxy$_1$).

```
(* private channels used to determine equality of strings *)
(* there is no XOR modeling for equal strings canceling each other out *)
(* leading to emulate PXH comparison in this way *)
free csid_str_map: channel [private].
free cpxh_sid_sid_map: channel [private].
```

Figure 5: Special channels used to emulate the PXH comparison due to the lack of XOR functionality support in ProVerif. These channels are private and are not made accessible to the adversary.

```
(* client: process clientCollection and process clientCollectionAttacker*)
out(csid_str_map, (sidstr, str));
(* aggregator: process aggregatorComparison *)
out(cpxh_sid_sid_map, (pxhA, (sid1, sid2)));
(* comparing proxy (in this model, proxy2): process proxy2Counting *)
in(cpxh_sid_sid_map, (=pxhA, (sid1c: bitstring, sid2c: bitstring)));
in(csid_str_map, (=sid1c, str1c: bitstring));
in(csid_str_map, (=sid2c, str2c: bitstring));
```

Figure 6: Use of the special channels in each process.

store. To perform a lookup of a key k, we use the statement $in(cds, (=k, v)); out(cds, (k, v))$; similar to Koi [23], in which cds is the private channel used as the datastore. This statement first retrieves the message using the conditional lookup, which removes the message from the channel and binds the variable v to the value of key k, and then adds the same message back again, such that it can be used for multiple lookups.

4.4 Unlinkability

We model unlinkability in our system similar to Koi [23] (shown in Figure 7). Any two variables that an adversarial component has access to at a single protocol step are explicitly linked using the private LINK function. The result of this function is made available to the adversary via the *spyAtt* channel in the model (§5.1). We use the query functionality of ProVerif to see whether the adversary has access to the explicit linking information about two variables.

The adversary also can use two public functions to infer the linkability of two variables. The INFER_SYMMETRY models the symmetry property of the LINK function: if the variables a is linked to b, then b is also linked to a. The INFER_TRANSITIVITY models the transitivity property of the LINK function: if a and b are linked to each other, and b and c are linked to each other, then a and c are also linked. Note that a and c may not have been accessible by the adversary at a single protocol step (e.g., collection of encrypted strings). The INFER_TRANSITIVITY function enables the adversary to infer linkability of such variables: a and b may be available at the collection of encrypted strings, and b and c may be available at the comparison and counting, and the adversary will still be able to link a and c together.

Alternative to this explicit LINK function, one can also make the LINK function public. This

```
fun LINK(bitstring, bitstring): bool [private].
reduc forall a: bitstring, b: bitstring; INFER_SYMMETRY(LINK(a, b)) = LINK(b, a).
reduc forall a: bitstring, b: bitstring, c: bitstring;
INFER_TRANSITIVITY(LINK(a,b), LINK(b,c)) = LINK(a,c).
```

Figure 7: Formal definition of linkability.

```
fun EXISTS(bitstring): bool [private].
```

Figure 8: Formal definition of existence. The result is made available to the adversary in the model.

approach essentially would allow the adversary to link *any* two variables it has access to, without needing the INFER_SYMMETRY and INFER_TRANSITIVITY functions to achieve the same link. However, this approach leads to the adversary being able to link any two variables at *any* time of the protocol steps, leading to many false attacks.

For example, assume the adversary is Proxy_1 and it runs its own clients. These clients' string types are naturally available to the adversary at Proxy_1. Proxy_1 also interfaces with honest clients, such that their network address is available. A public LINK function causes ProVerif to think that the adversary can link the honest client's address to the string type it knows from its own clients. As a result, many false attacks are reported by ProVerif.

4.5 Existence of a String

It is not possible to model counts in ProVerif. As a result, it is not straightforward to model an attack, in which the adversary creates enough Sybil clients and uses them to artificially inflate a string's count. This inflated count then be used in an attack, such as deducing the existence of a string value at a real client. To overcome this limitation, we consider the end goal of the adversary. We model a specific function that denotes the existence of a string value (Figure 8).

In our model, whenever the adversary at any component can deduce that a string value exists (i.e., by knowing the comparison result is equal and knowing one of the compared strings), an EXISTS(...) message is emitted on the public channel the adversary has access to (i.e., *spyAtt*). As a result, we can emulate the attacks where the adversary can exploit the comparison result and deduce that a string value exists at a real client.

5 Protocol Model and Verification Results

Here, we present the formal model of our system. For clarity, we describe the mirror operation 1, in which Proxy_2 is responsible for the comparison (Figure 9). In our model, we place the adversary separately at each component along our assumption that they are not going to collude. We then describe which variables are of interest to the adversary, which attacks are modeled, and any potential attacks ProVerif finds. We then reason about why some of these potential attacks are false attacks.

We divide our description along the lines of our protocol's phases. Each protocol phase builds on top of the previous one, such that the adversary can utilize the information it might have obtained

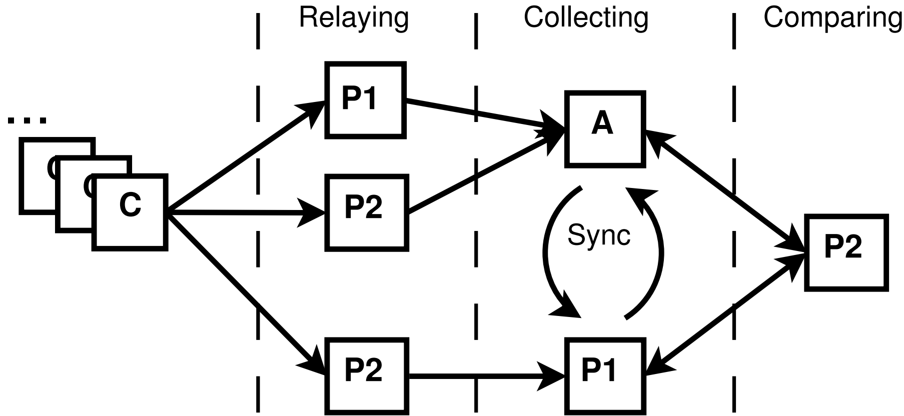

Figure 9: Mirror operation 1. The vertical, dashed lines separate the three roles.

Table 2: Attacks found by ProVerif by various adversaries. The attack in the last row is known, but cannot be modeled in ProVerif.

Adversary	Attack	Found by ProVerif?	Validity (Reasoning)
Aggregator with fake clients	LINK(strtype1, strtype2)	Yes	False (§5.3.1)
	LINK(str1, str2)	Yes	False (§5.4.1)
Proxy_1	access to client IP	Yes	False (§5.3.2)
	LINK(pipC, ipC)	Yes	False (§5.4.2)
Proxy_2	access to client IP	Yes	False (§5.3.2)
Proxy_2 with fake clients	EXISTS(str1) (known *sid* values)	Yes	True with incomplete protocol (§5.5.1)
	EXISTS(str1) (known PXH values)	Yes	True with incomplete protocol (§5.5.2)
	EXISTS(str1) (count-as-a-signature)	No	True with very low probability (§5.5.3)

during an earlier phase. For example, in the collection phase, the adversary still has access to the information that might have been exposed to the adversary during the initialization phase.

All model files along with documentation can be found at the following address: `https://www.mpi-sws.org/~iakkus/private/verif/`

5.1 Adversary Model

The standard adversary in ProVerif has access to public channels and can observe only messages that are sent on those channels. In our system, components interact with each other using encrypted channels that are modeled as private channels in ProVerif, preventing the adversary access to these variables. Other variables in compoents' internal states would not also be visible to the adversary. As a result, the standard adversary would not have access to any of the secret information in our protocol.

Our protocol, however, assumes that the components are honest-but-curious, meaning that they

```
(* attacker queries *)
(* adversary has access to the client IP address? (anonymity) *)
query attacker(ipC).

(* adversary can link the client to a string type? (unlinkability) *)
query attacker(LINK(ipC, strtype1)).
query attacker(LINK(ipC, strtype2)).

(* adversary can link a client to two string types? (anonymous profiling) *)
query attacker(LINK(strtype1, strtype2)).

(* adversary can link a client with a string? (unlinkability) *)
query attacker(LINK(ipC, str1)).
query attacker(LINK(ipC, str2)).
query attacker(LINK(ipC, str3)).

(* adversary can link a client to two strings? (anonymous profiling) *)
query attacker(LINK(str1, str2)).
query attacker(LINK(str1, str3)).

(* adversary can deduce that there exists a client with a string? (existence) *)
query attacker(EXISTS(str1)).

(* adversary can correlate pseudo IP address to a real IP address? *)
query attacker(LINK(pipC, ipC)).
```

Figure 10: List of items queried throughout our model to see whether the adversary has access to the respective piece of information.

will try to learn as much information about clients as possible from the variables they obtain. As a result, we need a way to let the ProVerif adversary have access to the internal state of the component, where the adversary is modeled to be.

To do so, we use a public channel (i.e., *spyAtt*) similar to Koi [23]. Depending on the component we model as the adversary (e.g., adversarial aggregator), we emit messages that contain the internal state of that component on the public channel.

5.2 Adversary Goals

Figure 10 shows the list of queried items throughout our model. Depending on which entity is the adversary, some queries will be trivial cases leading to false attacks. For example, querying whether the adversary at a proxy has access to the client IP does not make sense: one of the tasks of the proxy is to provide clients with network anonymity. Table 2 gives a summary of the attacks ProVerif finds and cannot find, whether the attacks it finds are false along with the section numbers explaining the reasoning.

5.3 Discovery Initialization

5.3.1 Adversary at the Aggregator

To distribute the string discovery parameters to the clients, the aggregator needs to learn the string type. However, this information by itself is not useful to the aggregator, because it already organizes all discovery procedures and knows the string types from analysts. Rather, the linkage between a client and its string types are of interest to the adversary. Additionally, the adversary at the aggregator may want to learn the string types a given client has, such that it can anonymously profile the client. At this phase, the clients do not send their strings yet.

ProVerif cannot find any attacks in the protocol as expected, because the proxies forward the client requests to the aggregator without exposing the client IP address. [**iea: Traffic analysis?**]

The aggregator may run its own clients, whose string types it knows. When ProVerif is run, it finds an attack in which the aggregator can anonymously profile a client: the adversary can obtain the linkage between two string types (e.g., LINK(String Type 1, String Type 2)). When we investigate the attack trace generated by ProVerif, we find that the adversary accesses this information from the clients it runs. In other words, the aggregator anonymously profiles its own clients! When the adversary's clients are not used, ProVerif cannot find any other attacks.

5.3.2 Adversary at Proxy_1 or Proxy_2

The proxies provide the clients with network anonymity, and thus, see the client address. However, the requests containing the string types of the client are XOR-encrypted, such that the proxies do not see them. Generic string types (e.g., 'visited websites') are available at each client, such that the adversary does not gain any new information. For analyst-specific strings, the XOR-encryption prevents each proxy from learning the string type assuming there is no collusion between the proxies. Similarly, when the proxies forward the string discovery parameters, they cannot obtain any information about the string types, because the parameters are also XOR-encrypted. ProVerif cannot find any attacks.

5.4 Collection of Encrypted Strings

5.4.1 Adversary at the Aggregator

During the collection of encrypted strings, the aggregator receives the string type. This string type is used to compile the comparison lists. For example, multiple analyst-specific string types are put into the same comparison list. An adversary at the aggregator may want to obtain the client IP and link it to the string type it receives. However, the client sends its string type over the proxies in a split form. Therefore, it is similar to the discovery initialization and the aggregator cannot obtain the linking between the clients and their string types. At this point, the client strings are still XOR-encrypted, the aggregator holding the R value and the other collecting component (e.g., Proxy_1) holding the matching X value.

The aggregator may run its own clients, whose string types and strings it knows. Again, ProVerif finds an attack, in which the adversary can obtain the linkage between two string types similar to the initialization phase, but also two strings (e.g., LINK(String 1, String 2)). The investigation of the attack trace again shows that the adversary accesses this information from the clients it runs. Without the adversary's clients, ProVerif cannot find any other attack.

5.4.2 Adversary at Proxy_1

Similar to the above description, ProVerif finds false attacks about anonymous profiling when the adversary runs its own clients to participate in string discovery procedures.

Proxy_1 forwards the double-split messages that carry the string type to the aggregator, and therefore, interacts with the client directly. However, it does not see the client IP for the collection of the X values and only receives a pseudo IP pIP assigned by Proxy_2. ProVerif finds an attack in which the adversary can correlate the client IP with the pseudo IP address, because it has access to both variables. This attack is a false attack, because the system assumes that there will be many clients participating in a string discovery procedure. Therefore, the probability of correlating the pIP value to the client IP will be inversely proportional to the number of cients.

This attack can also be interpreted as follows: Proxy_1 could run fake clients to generate traffic. It can keep a list of pIP values it receives and then filter its own clients by using the X values its clients have sent. Similarly, Proxy_1 can filter out the client IP addresses it sees when it forwards the double-split messages and remove its own clients' IP addresses from its list. Using both lists, it can try to reverse-engineer the correlation between the remaining client IP addresses and the remaining pIP values. However, in order for Proxy_1 to succeed, there should only be traffic from the target client and Proxy_1's clients. If there are other clients that are not controlled by Proxy_1, then this attack's success probability is lowered with every extra client. The system assumes that there are many clients participating, such that the number of Proxy_1's clients is less than the other clients.

The adversary sees the client IP address, because it is the proxy's task to provide the clients with network anonymity. Besides this trivial case, ProVerif cannot find any other attacks.

5.4.3 Adversary at Proxy_2

Proxy_2 assigns a pseudo IP address to each client and forwards the X values to Proxy_1. Besides the false attacks about anonymous profiling and a trivial case of Proxy_2 accessing the client IP address, ProVerif cannot find any other attacks.

5.5 Blind Comparison and Counting - Adversary at Proxy$_2$

The comparison and counting of the encrypted strings is done blindly using the comparison of PXH values. This comparison does not leak any information about the strings being compared, except for their equality or inequality. This comparison result is not learned by the collecting components (i.e., Proxy$_1$ and the aggregator), and they do not receive any new piece of information regarding strings and string types (besides the information they obtained in the previous stages of the protocol). ProVerif finds the same (false) attacks described above when the adversary is considered to be one of these components. For this reason, we do not explicitly describe these cases.

On the other hand, Proxy$_2$ compares the PXH values and determines the equality of the strings. It puts equal strings into equality lists, such that if two strings are found to be equal, they are put into the same list. At the end of the counting process, Proxy$_2$ adds noise to each list's length length and filters the lists that are below the threshold. It then selects a representative string from each list, and reports the count to the aggregator.

To perform these tasks, Proxy$_2$ learns the comparison result. Proxy$_2$'s role to make the comparison enables it to deduce the existence of a string (besides the other pieces of information it may want to learn), if the necessary conditions arise. In the rest of this section, we describe more specific attacks involving the adversary at Proxy$_2$. We introduce bugs to the original protocol and let ProVerif find the attacks, and explain how the original protocol prevents these attacks. We finally describe another attack, which ProVerif cannot find due to its limitation to model counts and how this attack is not of concern because of the system's underlying assumptions.

5.5.1 Known *sid* Values Attack

Proxy$_2$ may run clients and send known strings. It can determine that one of these strings is being compared with another string by utilizing the sid values the clients send and the collecting components (i.e., Proxy$_1$ and the aggregator) use as identifiers for the compared strings. We model this attack and verify that Proxy$_2$ indeed can use this approach to determine the existence of a string: ProVerif finds the attack and generates the attack trace.

Akkus et al.'s original protocol prevents this attack by modifying the original sid values by overwriting them with a shared secret R_s between the collecting components, such that $sid'_i = H(sid_i \oplus R_s)$. When the original protocol is modeled, ProVerif cannot find any other attacks.

5.5.2 Known R Values Attack

Another method Proxy$_2$ can utilize to determine that it is comparing a known string value with another unknown string value is to use the R values.[3] When the PXH operation is applied without the secret shared between the collecting components, it is possible for Proxy$_2$ to identify $\mathrm{PXH}_{Aggregator}(sid'_i, sid'_j) = H(R_1 \oplus R_2)$. [4] Consequently, when a known PXH value is received by Proxy$_2$, it can deduce that sid'_i corresponds to sid_i and sid'_j corresponds to sid_j (or vice versa). Afterwards, when an unknown string is compared with one of these identified strings, Proxy$_2$ will be able to deduce the existence of a string.

Indeed, if the PXH values are computed without the secret between the collecting components, ProVerif finds the attack trace, in which the adversary at Proxy$_2$ can launch this attack. The

[3] Or X values.

[4] Or $\mathrm{PXH}_{Proxy_1}(sid'_i, sid'_j) = H(X_1 \oplus X_2)$.

original protocol modifies the PXH values with the secret, such that $\mathrm{PXH}_{Aggregator}(sid'_i, sid'_j) = H(R_1 \oplus R_2 \oplus R_s)$ and $\mathrm{PXH}_{Proxy_1}(sid'_i, sid'_j) = H(X_1 \oplus X_2 \oplus R_s)$. After this correction, ProVerif cannot find any additional attacks.

5.5.3 Count-as-a-Signature Attack

Although we can model the above attacks and ProVerif is able to find them, there is another method for Proxy_2 to identify strings its clients sent: by using the count of strings as a signature. Proxy_2 can run clients to send a particular string. When the equality lists are formed, Proxy_2 can identify these strings from the list's length and identify their sid' values. Afterwards, Proxy_2 can utilize the comparison results of these strings with other unknown strings to deduce a string's existence.

Unfortunately, due to the lack of count support, this attack cannot be modeled in ProVerif. As a workaround, one could manually create a certain number of string instances; however, one cannot check the number of instances against a constant value.

This attack, however, is not a concern for the system in practice for the following reasons. The system assumes that the string distributions most probably will follow power law. This assumption means that there will be a long tail in the distribution, leading to many strings having small counts. As a result, to create a unique signature of string injected via clients, Proxy_2 would have to create many clients, increasing the difficulty of this attack, with the duplicate detection forcing Proxy_2 to use one client for one string instance. Furthermore, Proxy_2 would have to guess the number of actual clients with that string to correctly estimate and identify the count as the signature. We think that in practice, this attack will not be feasible with very high probability.

5.6 Duplicate Detection

To prevent a malicious client from arbitrarily manipulating string counts by sending the same string multiple times, our system performs a duplicate detection before it counts the encrypted distinct strings. The high-level idea is to run the same blind comparison protocol, but this time among all strings from a given client: equal strings will be duplicates, indicating a malicious client without revealing any strings. This phase is similar to the comparison and counting phase, in which Proxy_1 and the aggregator perform the PXH operations and Proxy_2 compares the PXH values.

Similar to the comparison and counting phase, ProVerif finds the same (false) attacks when the adversary is considered to be either the aggregator or Proxy_1. We do not discuss these cases any further.

As for the adversary at Proxy_2, it still learns the comparison result. However, this result is not exploitable by the adversary, because the comparison is performed only among the strings from the same client. That means, if Proxy_2 were to use fake clients and send strings, they would be compared with each other (and not with other honest clients' strings). As a result, Proxy_2 cannot expoloit the comparison result to deduce the existence of a rare string value at an honest client.

Additionally, Proxy_1 and the aggregator use different secrets in the duplicate detection and comparison phases to modify the sid and PXH values. As a result, Proxy_2 cannot correlate the strings it compares in both of these phases.

6 Limitations of our Formal Model

Here we describe some limitations of our formal model. We also reason about why these limitations do not affect our privacy goals in practice.

6.1 Modeling Full XOR functionality

The exclusive-OR (XOR) operation is commutative and associative. In addition, equal strings cancel each other out when XORed together. These properties cannot be modeled in ProVerif explicitly. As a result, certain attacks making use of these properties cannot be explored by ProVerif. Here, we describe how some of our model's properties allow us partly abstract away some of the functionality requirements.

Our model utilizes channels as datastores as well as for end-to-end private communications between components. Channels in ProVerif are asynchronous and messages sent in the channels can be retrieved in any order. By retrieving any two messages in different orders (either from a datastore or a communication channel), ProVerif can 'emulate' the commutativity property.

Our protocol utilizes the cancellation property of XOR. To overcome this lack of functionality, we use special private channels (i.e., not accessible to the adversary) and emulate the PXH comparison operation: The PXH_{P_1} and PXH_A values are computed using client split messages and a secret by Proxy_1 and the aggregator, which are then used to look up client strings and compare them. As a result, the comparison operation can be emulated.

Although the use of these special channels allow us to work around the lack of cancellation property of XOR, it also weakens the ProVerif attacker: the attacker cannot arbitrarily XOR any two split messages it has access to, and maybe discover secrets. A similar argument is also true for the associativity property.

We acknowledge these weaknesses in our model, but also point out that our protocol does not trust any one component to have both split messages, such that one component cannot obtain the original strings or string types (i.e., X values are held by Proxy_1 and R values are held by the aggregator). When there is no collusion among these components, which we assume, the adversary cannot access both values at the same time. Furthermore, the R values are generated independently for each string. As a result, XORing any two split messages will not yield anything meaningful. This case is similar to one component holding the random keys for symmetrically encrypted strings and the other holding the encrypted strings. Our system also does not depend on the associativity property for the XOR-encryption.

6.2 Modeling the Noisy threshold

Our model does not consider the threshold and the noise that is added to the counts of strings. This noise is essential to ensure that an adversary cannot make the system reveal a string value whose count is artificially inflated (i.e., via fake clients) to be above the threshold. We cannot use counts in ProVerif, such that we cannot model the threshold nor the noise. Our protocol depends on the Laplace noise that is used by differential privacy [20] to prevent this attack.

A computation, C, provides ϵ-differential privacy [20–22] if the following inequality is satisfied for all datasets, D_1 and D_2, that differ on one record, and for all outputs $O \subseteq Range(C)$:

$$\Pr[C(D_1) \in O] \leq \exp(\epsilon) \times \Pr[C(D_2) \in O] \tag{1}$$

Namely, the probability of a computation producing a given output is almost independent of the existence of any individual record in the dataset. This property is achieved by adding noise to the computation's output with the privacy parameter ϵ: the smaller ϵ, the more privacy.

In our setting, this property suggests the following: If there is a string with t-1 Sybils, the probability of the string being discovered (and decrypted) is almost independent of any honest client with that string. In other words, whether a real client with that string exists does not significantly affect the discovery: The client may exist (i.e., the noise-free count is t) and the noise may be negative, and thus, the string may not be discovered. On the other hand, the client may not exist (i.e., the noise-free count is t-1) and the noise may be positive, and thus, the string may be discovered. These two cases are indistinguishable.

In our model, we abstract away the threshold and model the adversary's end goal of deducing the existence of a string by exploiting the comparison result. In our system, this goal is only achievable by exploiting the comparison result between an unknown string and a known string sent by a fake client. In our system, this goal is only achievable by Proxy_2. Our ProVerif model then covers the cases, in which Proxy_2 can identify its strings and ensures that our protocol prevents this identification. We then reason about a case that cannot be modeled in ProVerif and reason why it is not a problem in practice according to our assumptions. As a result, the comparison result cannot be exploited by Proxy_2 (§5.5.3).

6.3 Modeling the Sample-Identify-Count-Filter Optimization

Our Sample-Identify-Filter-Count (SICF) optimization requires the model of counting support: the strings in each sample need to be counted, and the most popular strings are requested to compared with the rest of the strings to obtain a full count. For this reason, we did not consider this optimization in our model.

The reasoning about the privacy of this optimization is the following: To filter equal strings from the comparison list, Proxy_1 and the aggregator learn comparison results between some strings. If they identify one of these strings (e.g., their fake clients sent it), they can expose honest clients' strings that are equal to the identified string. However, they learn *many* sid' values of strings equal to *any* one of the p common strings, and thus, cannot be certain which strings are actually equal. In addition, these results belong to the p most common strings in the *random* sample, which reflects the string counts in the original comparison list: to expose a rare string, an adversary would need to send it so many times to make it one of the p most common strings in the sample. Our duplicate detection raises the bar for the adversary, forcing it to use more Sybils.

6.4 Modeling the Short Hashes Optimization

It is straightforward to model our optimization that uses a small number of hash values. The hash value of the string only needs to be transmitted during the collection of the encrypted strings to the aggregator, similar to the string type. One can simply imagine that the string type already encodes the hash value. For example, one string type could be 'websites_with_hash_0', while another could be 'websites_with_hash_1', 'websites_with_hash_2' and so on.

The reasoning about the privacy of this optimization is the following: With a small number of hash buckets, many distinct string values will map to the same bucket (i.e., many hash collisions). Thus, the information gained about a string by knowing its bucket will be small. For example, with 128 buckets, the average numbers of distinct string values per bucket in our datasets are about 7.8K

for websites and 101K for search phrases [13]. Clients and watchdogs can set a maximum value for the number of buckets allowed (e.g., $\leq$128).

7 Conclusion

We have formally modeled and verified many aspects of the privacy-preserving string discovery system proposed in [13] using ProVerif [7], including anonymity and unlinkability according to the definitions by Pfitzmann and Köhntopp [28] as well as the existence of a rare string. In our model, we placed the adversary at each component and made the component's internal variables accessible to the adversary, modeling honest-but-curious components. The adversary also had the ability to run fake clients. We reasoned about the potential attacks found by ProVerif and showed that they are false. We also modified the original protocol to introduce bugs and showed that ProVerif can find the attacks, showing the original protocol is effective against these attacks. Additionally, we describe an attack that cannot be modeled in ProVerif and explain why that attack is not an issue in practice according to the assumptions made by the system. Finally, we discussed the limitations of our model, and the parts of the system (e.g., the noisy threshold, the SICF optimization) that cannot be modeled in ProVerif due to its limitations.

Acknowlegdments

We thank Jan-Oliver Kaiser for his comments on this report. We also thank Deepak Garg for his feedback and suggestions on how to improve our model, and for his comments on this report.

References

[1] BlueKai Consumers. http://bluekai.com/consumers_optout.php.

[2] ComScore: Mobile Will Force Desktop Into Its Twilight In 2014. http://www.businessinsider.com/mobile-will-eclipse-desktop-by-2014-2012-6.

[3] FTC Issues Final Commission Report on Protecting Consumer Privacy. http://ftc.gov/opa/2012/03/privacyframework.shtm.

[4] Internet Access Statistics. http://www.ons.gov.uk/ons/dcp171778_301822.pdf.

[5] Lawsuit accuses comScore of extensive privacy violations. http://www.computerworld.com/s/article/9219444/.

[6] Privacy Lawsuit Targets Net Giants Over 'Zombie' Cookies. http://www.wired.com/threatlevel/2010/07/zombie-cookies-lawsuit.

[7] ProVerif. http://proverif.inria.fr/.

[8] Quantcast Clearspring Flash Cookie Class Action Settlement. http://www.topclassactions.com/lawsuit-settlements/lawsuit-news/920.

[9] Quantcast Opt-Out. http://www.quantcast.com/opt-out.

[10] The Do Not Track Option: Giving Consumers a Choice. `http://ftc.gov/opa/reporter/privacy/donottrack.shtml`.

[11] Web Tracking Protection. `http://www.w3.org/Submission/web-tracking-protection/`.

[12] Abadi, M., and Fournet, C. Mobile Values, New Names, and Secure Communication. In *POPL* (2001).

[13] Akkus, I. E., Chen, R., and Francis, P. String Discovery for Private Analytics. In *Max Planck Institute for Software Systems Technical Report MPI-SWS-2013-006* (2013).

[14] Akkus, I. E., Chen, R., Hardt, M., Francis, P., and Gehrke, J. Non-tracking web analytics. In *CCS* (2012).

[15] Applebaum, B., Ringberg, H., Freedman, M. J., Caesar, M., and Rexford, J. Collaborative, Privacy-preserving Data Aggregation at Scale. In *PETS* (2010).

[16] Burkhart, M., Strasser, M., Many, D., and Dimitropoulos, X. A. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium* (2010).

[17] Chen, R., Akkus, I. E., and Francis, P. SplitX: High-Performance Private Analytics. In *SIGCOMM* (2013).

[18] Chen, R., Reznichenko, A., Francis, P., and Gehrke, J. Towards Statistical Queries over Distributed Private User Data. In *NSDI* (2012).

[19] Duan, Y., Canny, J., and Zhan, J. Z. P4P: Practical Large-Scale Privacy-Preserving Distributed Computation Robust against Malicious Users. In *USENIX Security Symposium* (2010).

[20] Dwork, C. Differential Privacy. In *ICALP* (2006).

[21] Dwork, C. Differential Privacy: A Survey of Results. In *TAMC* (2008).

[22] Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC* (2006).

[23] Guha, S., Jain, M., and Padmanabhan, V. N. Koi: A location-privacy platform for smartphone apps. In *NSDI* (2012).

[24] Hardt, M., and Nath, S. Privacy-aware personalization for mobile advertising. In *CCS* (2012).

[25] Küsters, R., and Truderung, T. Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach. In *CCS* (2008).

[26] Lee, S., Wong, E. L., Goel, D., Dahlin, M., and Shmatikov, V. πbox: a platform for privacy-preserving apps. In *NSDI* (2013).

[27] Milner, R. *Communicating and mobile systems: the pi calculus.* Cambridge university press, 1999.

[28] Pfitzmann, A., and Koehntopp, M. Anonymity, Unobservability, and Pseudonymity - A Proposal for Terminology. In *Designing Privacy Enhancing Technologies*. 2001.

[29] Rabin, M. O. How to exchange secrets by oblivious transfer. Tech. rep., TR-81, Harvard Aiken Computation Laboratory, 1981.

[30] Roesner, F., Kohno, T., and Wetherall, D. Detecting and Defending Against Third-Party Tracking on the Web. In *NSDI* (2012).