8-1-2011

Differential Privacy Under Fire

Andreas Haeberlen
University of Pennsylvania, ahae@cis.upenn.edu

Benjamin C. Pierce
University of Pennsylvania, bcpierce@cis.upenn.edu

Arjun Narayan
University of Pennsylvania

Differential Privacy Under Fire

Andreas Haeberlen Benjamin C. Pierce Arjun Narayan
University of Pennsylvania

Abstract

Anonymizing private data before release is not enough to reliably protect privacy, as Netflix and AOL have learned to their cost. Recent research on *differential privacy* opens a way to obtain robust, provable privacy guarantees, and systems like PINQ and Airavat now offer convenient frameworks for processing arbitrary user-specified queries in a differentially private way. However, these systems are vulnerable to a variety of covert-channel attacks that can be exploited by an adversarial querier.

We describe several different kinds of attacks, all feasible in PINQ and some in Airavat. We discuss the space of possible countermeasures, and we present a detailed design for one specific solution, based on a new primitive we call *predictable transactions* and a simple differentially private programming language. Our evaluation, which relies on a proof-of-concept implementation based on the Caml Light runtime, shows that our design is effective against remotely exploitable covert channels, at the expense of a higher query completion time.

1 Introduction

Privacy is a problem. Vast amounts of data about individuals is constantly accumulating in various databases—patient records, content and link graphs of social networks, mobility traces in cellular networks, book and movie ratings, etc.—and there are many socially valuable uses to which it can potentially be put. But, as Netflix and others have discovered [3, 22], even when data collectors *try* to protect the privacy of their customers by releasing anonymized or aggregated data, this data often reveals much more than intended, especially when it is combined with other data sources. To reliably prevent such privacy violations, we need to replace current ad-hoc solutions with a principled data release mechanism that offers strong, provable privacy guarantees.

Recent research on *differential privacy* [8–10] has brought us a big step closer to achieving this goal. Differential privacy allows us to reason formally about what an adversary could learn from released data, while avoiding many assumptions (e.g., what exactly the adversary might try to learn, or what he or she might already know) that have been the cause of privacy violations in the past. Early work on differentially private data analysis relied on manual proofs by privacy experts that the answers to particular queries were safe to release [21]; today, systems like PINQ [20] and Airavat [26] can perform differentially private data analysis automatically, without needing a human expert in the loop.

Airavat and PINQ go beyond just certifying queries by the data owner as differentially private; they are explicitly designed to support *untrusted* queries over private databases. In this model, a third party is permitted to submit arbitrary queries over the database, but the data owner imposes a "privacy budget" that limits the amount of information the third party can obtain about any individual whose data is in the database. The system analyzes each new query to determine its potential "privacy cost" and allows it to run only if the remaining balance on the privacy budget is sufficiently high. This mode of operation is attractive for many scenarios; for example, Netflix could give researchers access to its database of movie ratings via such a query interface and still give strong privacy assurances to customers. An adversarial querier could not, for instance, obtain an accurate answer to the query "*Has John Doe watched any adult movies?*" because the cost of such a query would exceed any reasonable privacy budget.

However, Airavat and PINQ both contain vulnerabilities that can be exploited by an adversary to extract private information through covert channels.[1] The reason is that these systems rely on the assumption that the querier can observe *only* the result of the query, and nothing else. In practice, however, the querier is also able to observe other effects of his query, such the time it takes to com-

[1] The designers of these systems were aware of these covert channels, and each addresses them to some extent. See Sections 3.5 and 3.6.

plete. Such observations can be exploited to mount a covert-channel attack. To continue with our earlier example, the adversary might run a query that always returns zero as its result but that takes one hour to complete if John Doe has watched adult movies and less than a second otherwise. Both Airavat and PINQ would consider the output of such a query to be safe because it does not depend on the contents of the private database at all. However, the adversary can still learn with perfect certainty whether John Doe has watched adult movies—a blatant violation of differential privacy. PINQ's prototype implementation also permits global variables to be used as covert channels to leak private information during query execution.

Covert channels have plagued computer systems for many years [1, 2, 15, 16, 18, 27, 30, etc.], and they are notoriously difficult to avoid [7]. However, they are particularly devastating in a system that is designed to enforce differential privacy: if a channel allows the adversary to learn even a single bit of private information, the differential privacy guarantees are already broken! Thus, differential privacy puts particularly high demands on a defense against covert channels; merely limiting the bandwidth of the channels is not enough.

Fortunately, the untrusted-query scenario has two features that make a solution feasible. First, there is no need to allow the querier direct access to the machine that hosts the database; he can be forced to submit queries and receive results over the network. This rules out difficult channels such as power consumption [17] and electromagnetic radiation [13, 24], essentially leaving the adversary with just two channels: the privacy budget and the query completion time.

Our key insight is that, in this specific scenario, these two channels can be closed *completely* through a combination of two techniques. The budget channel can be closed by using program analysis to statically determine the privacy cost of each query. Thus, the deduction from the privacy budget is independent of the database contents. The external timing channel can be closed by a) breaking each query into "microqueries" that operate on a single database row at a time, and by b) enforcing that each microquery takes a fixed amount of time. (If necessary, the microquery is aborted and a *default value* is returned. In the context of differential privacy, this is safe—and does not open another channel—because the privacy cost of the default values is already included in the privacy cost of the query.) Thus, we can obtain strong privacy assurances even if the adversary can pose arbitrary queries and can observe all the (remotely measurable) channels that are possible in our model.

We present the design of Fuzz, a system that implements this defense. Fuzz uses a novel type system [25] to statically infer the privacy cost of arbitrary queries written in a special programming language, and it uses a novel primitive called *predictable transactions* to ensure that a potentially adversarial computation completes within a specific time or returns a default value. We have built and evaluated a proof-of-concept implementation of Fuzz based on the Caml Light runtime system [5, 19]. Our results show that Fuzz effectively closes all known remotely exploitable channels, at the expense of a higher query completion time.

Implementing predictable transactions is challenging in practice: Fuzz must be able to abort an arbitrary and potentially adversarial computation by a specified deadline, even if the adversary is actively trying to cause the deadline to be missed, and must ensure that—whether or not the computation is aborted—it leaves no lingering traces that can measurably affect the program's overall execution time (garbage in the heap, VM pages that must later be freed by the OS, etc). Nevertheless, we show that, across a variety of adversarial queries that exploit different attack strategies, our implementation exhibits extremely small variation in completion time—on the order of the time required to handle a single timer interrupt. This variation is so small that it is difficult to measure even on the machine itself. Thus, it would be useless to a remote attacker, who would have to measure it across a wide-area network using the limited number of trials that the privacy budget permits.
In summary, we make the following contributions:

1. a detailed analysis of several classes of covert-channel attacks and a discussion of which are feasible in PINQ and Airavat (Section 3);
2. an analysis of the space of potential solutions (4);
3. a concrete design for one specific solution, based on default values and predictable transactions (5+6);
4. a proof-of-concept implementation of our design (7); and
5. an experimental evaluation (8).

We close with a discussion of related work and a few concluding thoughts.

2 Background

Before describing our attacks and the Fuzz design and implementation, we briefly review some technical background on differential privacy, function sensitivity, and differentially private programming languages.

2.1 Differential privacy

Differential privacy [8] is a property of randomized functions that take a database as input and return a result that is typically some form of aggregate (a real number representing a count; a histogram; etc.). The database (db)

is a collection of "rows," one for each individual whose privacy we mean to protect.

Informally, a randomized function is differentially private if arbitrary changes to a single individual's row (keeping other rows constant) result in only statistically insignificant changes in the function's output distribution; thus, any individual's presence in the database has a statistically negligible effect. Formally [12], differential privacy is parametrized by a real number ε, corresponding to the strength of the privacy guarantee: smaller ε's yield more privacy. Two databases b and b' are considered *similar*, written $b \sim b'$, if they differ in only one row. We then say that a randomized function $q : \mathsf{db} \to \mathbb{R}$ is *ε-differentially private* if, for all possible sets of outputs $S \subseteq \mathbb{R}$, and for all similar databases b, b', we have $Pr[q(b) \in S] \leq e^{\varepsilon} \cdot Pr[q(b') \in S]$. That is, when the input database is changed in one row, there is at most a very small multiplicative difference (e^{ε}) in the probability of *any* set of outcomes S.

Methods for achieving differential privacy can be attractively simple—e.g., perturbing the true answer to a numeric query with carefully calibrated random noise. For example, the query *"How many patients at this hospital are over the age of 40?"* is intuitively "almost safe": safe because it aggregates many individuals' information together, but only "almost" because, if an adversary happened to know the ages of every patient except John Doe, then answering this query exactly would give him certain knowledge of a fact about John. The differential privacy methodology rests on the observation that, if we add a small amount of random noise to this query's result, we still get a useful estimate of the true answer while obscuring the age of any single individual. By contrast, the query *"How many patients named John Doe are over the age of 40"* is plainly problematic, since the answer is very sensitive to the presence or absence of a single individual. Such a query cannot usefully be privatized: if we add enough noise to mostly obscure the contribution of John Doe's age, there will be essentially no signal left.

2.2 Compositionality and privacy budgets

An important consequence of the definition of differential privacy is that composing a differentially private function with any other function that does not, itself, depend on the database yields a function that is again differentially private—that is, no amount of postprocessing, even with unknown auxiliary information, can lessen the differential privacy guarantee. This allows us to reason about harmful effects of data release that might seem quite far removed from the function that is actually being computed.

Another important property of differential privacy is that its guarantee degrades gracefully under repeated application: a pair of two ε-differentially private functions is always 2ε-differentially private, when taken together. This allows us to think of having a fixed "privacy budget" up front, which is slowly exhausted as queries are answered: if our privacy budget is ε, we may feel free to independently answer k queries, where the i^{th} query is ε_i-differentially private and $\sum_i \varepsilon_i \leq \varepsilon$, without fear that the aggregation of these k queries will violate ε-differential privacy.

2.3 Function sensitivity

The central idea in proofs of differential privacy is to bound the *sensitivity* of queries to small changes in their inputs. Sensitivity is a kind of continuity property; a function of low sensitivity maps nearby inputs to nearby outputs.

Sensitivity is relevant to differential privacy because the amount of noise required to make a deterministic query differentially private is proportional to its sensitivity. For example, the sensitivity of the two age queries discussed above is 1: adding or removing one patient's records from the hospital database can change the true value of each query by at most 1. This means that we should add the *same* amount of noise to *"How many patients at this hospital are over the age of 40?"* as to *"How many patients named John Doe are over the age of 40?"* This may appear counter-intuitive, but it achieves the right goal: the privacy of single individuals is protected to exactly the same degree in both cases. What differs is the *usefulness* of the results: knowing the answer to the first query with, say, a typical error margin of ± 100 could still be valuable if there are thousands of patients, whereas knowing the answer to the second query (which can only be zero or one) ± 100 is useless. We might try making the second query more useful by scaling its answer up numerically: *"Is John Doe over 40? If yes, then 1,000, else 0."* But this scaled query now has a sensitivity of 1,000, not 1, and so 1,000 times the noise must be added, blocking our attempt to violate privacy.

2.4 Programming with privacy

Early work on differential privacy has mostly focused on specific algorithms rather than general, compositional mechanisms: given a particular algorithm, we prove by hand that it is differentially private. Most of the time, this does not require much ingenuity—just applying known techniques—but even so, this approach doesn't scale well because it demands that each new algorithm be certified by a skilled, trusted human. A better approach is to automate this certification process with a programming language in which every well-typed program is guaranteed to be differentially private. Then (untrusted) non-experts can write as many different algorithms as they like, and the database administrator can rely on the language to ensure that privacy is not being violated.

Systems are beginning to be available that implement such languages—notably Privacy Integrated Queries (PINQ) [20] and Airavat [26]. PINQ is an embedded extension of C# that tracks the privacy impact of variety of relational algebra operations on database tables, as well as certain forms of query composition. Airavat integrates differential privacy into a distributed, Java-based MapReduce framework.

2.5 Processing model

Although PINQ and Airavat differ in many particulars, they embody essentially the same basic processing model, which we also follow in the Fuzz system described below. A *query* in each of these systems can be viewed as consisting of one or more *mapping* operations that process individual records in the database, together with some *reducing* code that combines the results of the mapping operations without directly looking at the database. When a query is submitted, the system verifies that it is ε_i-differentially private, deducts ε_i from the total privacy budget ε associated with the database, and—if ε remains nonzero—returns the query result. (Note that, in this model, we account for the possibility of collusion between adversaries by associating the privacy budget with the *database* and not with individual queriers. Thus, once the budget is exhausted, we must throw away the database and never answer any more queries.) We call the mapping operations *microqueries* and the rest of the code the *macroquery*.

Airavat implements a simple version of this model: a query consists of a sequence of chained microqueries ("mappers" in Airavat terminology) plus a selection from among a fixed set of macroqueries ("reducers"). The mappers are the only untrusted code: the reducers are part of the trusted base. When a query is submitted, the adversary must also declare the expected numerical range of its outputs, which amounts (since its input is a single record of the database) to stating its sensitivity. If the actual output ever falls outside of the declared range, it is clipped—in essence, the declared sensitivity is enforced by the system. From the declared sensitivity, Airavat can calculate how much noise must be added to the reducer's results to achieve ε-differential privacy.

In PINQ, macroqueries are written in LINQ, a SQL-like declarative language, which can be embedded in otherwise unconstrained C# programs. Microqueries can be general C# computations (optionally constrained by a checker method called `Purify`; see Section 3.5).

3 Attacks on differential privacy

Naturally, database administrators may be nervous about offering adversaries the opportunity to run arbitrary queries against their raw data. They will need strong assurances that such adversarial queries not only play by the rules of differential privacy but also have no *indirect* means of improperly leaking private information about individuals in the database. Unfortunately, this is not currently the case: while the authors of both PINQ and Airavat have anticipated the possibility of covert-channel attacks and have implemented either a partial defense (Airavat) or hooks for adding one (PINQ), both systems remain vulnerable to a range of attacks, as we now demonstrate.

3.1 Threat model

It is well known that covert channels are essentially impossible to eliminate if we allow the adversary to run other processes on the same computer that runs the query. Even if these other processes have no access to the database and cannot communicate directly with the query process, there are just too many ways for the query process to perturb local conditions in ways that can be measured fairly accurately if the observer is this close—e.g., processor usage, disk activity, cache pollution, etc. However, if we assume that the adversary is on the other end of a network connection, we have a much better chance of success. This is fortunate, since the demands of the situation are very strong. It is not enough to limit leakage to a low bandwidth or a small number of bits: even *one bit* is too much if that bit is the answer to *Does John Doe watch adult movies?*

We therefore assume that the database and associated query system are hosted on a private, secure machine. The adversary does not have physical access to this machine or its immediate environment (so that there is no way to measure its power usage, etc.) and can only communicate with it over a network. The adversary submits arbitrary queries to the system over the network. The system executes each query (if it determines that doing so is safe) and returns the answer over the network. The system also maintains a privacy budget for the database as a whole, and it refuses to answer any more queries once the budget is exhausted.

This threat model is shared by all differentially private query systems (PINQ, Airavat, and our Fuzz system), and its assumptions seem reasonable in practice. Essentially, it gives the adversary three pieces of information: (1) the actual answer to their query (a number, histogram, etc.), if any, (2) the *time* that the response arrives on their end of the network connection, and (3) the system's decision whether to execute their query or refuse because doing so would exceed the available privacy budget. However, this threat model still provides plenty of room for attacks on privacy. We will see that, unless appropriate steps are taken, both the decision whether or not to execute a query and the execution time itself can be used as channels to leak private information. In essence, both the query's finishing time and the fact that it is accepted

```
noisy sum, foreach r in db, of {
  if embarrassing(r)
    then { pause for 1 second };
  return 0
}
```

Figure 1: Timing attack example

or refused are *results* that the system is giving back to the adversary, and we need to consider whether the combination of *all* results—not just the query's numerical answer—is differentially private. Moreover, we will see, for PINQ, some ways that a malicious query may cause the actual *answer* to not be differentially private.

3.2 Timing attacks

Under the constraints of the above threat model, the easiest way for a query to send a bit to the adversary is by simply pausing for a long time (by entering an infinite loop, computing factorial of a million, etc.) when a certain condition is detected in the private data, as illustrated (in PINQ-like pseudocode) in Figure 1. The macroquery adds together the results of running the microquery on each row of the database (always 0) and finally adds some random noise to the total. Since almost all of the microquery instances finish very quickly, the distribution of query execution times observed by the adversary will change significantly when an embarrassing record exists in the database—a violation of differential privacy.

A simple "microquery timeout" will not solve this problem, for at least two reasons. First, the adversary can also signal the condition by causing the query to take an unusually *small* amount of time. The simple way to do this is to create an exception condition that aborts the entire query. If this is blocked (e.g., by trapping an exception in a microquery and replacing it with a default result just for that single microquery), the adversary can instead make all microqueries take a uniformly longish time (say, exactly two milliseconds) except when they detect the condition, in which case they terminate immediately. If the adversary happens to know exactly how many records are in the database, this leaks one bit. Second, the adversary can defeat a simple "microquery timeout" by causing side-effects in the microquery that will slow down the macroquery or other microqueries—for example, by allocating lots of memory to trigger garbage collection in the macroquery. We discuss this issue in more detail below.

3.3 State attacks

A different class of attacks involves using a channel *between* microqueries, such as a global variable, to break differential privacy of the result, as illustrated in Figure 2.

```
found = false;
noisy sum, foreach r in db, of {
  if (found) then { return 1 }
  if embarrassing(r) then {
    found = true;
    return 1
  } else { return 0 }
}
```

Figure 2: State attack example

```
noisy sum, foreach r in db, of {
  if embarrassing(r) then {
    run sub-query that uses
      a lot of privacy budget
  } else {
    return 0
  }
}
```

Figure 3: Privacy budget attack example

This time, the result of each microquery is either 0 or 1, depending on whether *any previous* microquery detected an embarrassing record. Since, in general, the embarrassing record will not be the last one in the database, this greatly magnifies the contribution of this one record to the result, again violating differential privacy.

3.4 Privacy budget attack

A related form of attack uses the query processor's decision whether to publicize the result of a query as a channel for leaking private data, relying on the fact that this decision can be influenced by actions of the query that in turn depend on private data. This idea can be applied to systems that use a *dynamic* analysis to determine the 'privacy cost' of a query, i.e., the amount that must be subtracted from the privacy budget before the result can be returned to the querier. As illustrated in Figure 3, the attack consists of looking for an embarrassing record and, when it is found, invoking some sub-query that will use up a bit of the remaining privacy budget. Once the outer query returns, the adversary simply checks how much the privacy budget has decreased.

3.5 Case study: PINQ

We have verified that the current PINQ implementation (version 0.1.1, released 08/18/09, available from [23]) is vulnerable to all of the above attacks. To demonstrate the vulnerabilities, we have written three example programs, each based on the test harness that comes with PINQ.

The original test harness computes several differentially private statistics on a given text file, including the

	Constant execution time Database size public ε-differential privacy	**Variable execution time** Database size private (ε, δ)-differential privacy
Static enforcement	Exact timing analysis	Time bound analysis Time noise
Dynamic enforcement	Timeouts Rounding up	Timeouts Time noise

Table 1: Four approaches to the timing-channel problem.

number of lines that contain a semicolon. When the program starts, it first reads the text file and creates a database whose rows each contain one line of text. Then it selects all the rows that contain a semicolon, using microqueries with a boolean predicate p, and finally performs a noisy count on the resulting set of rows.

Our attacks are implemented by changing the predicate p so that it produces some observable side-effects when the input file contains a certain string s. For the timing attack, we changed p so that, when invoked on a line that contains s, p performs an expensive computation that takes several seconds and cannot be optimized out. For the state attack, we added a static variable that is incremented by p when it discovers s, and we write the (un-noised) value of this variable to the console at the end. For the budget attack, we added a different static variable that contains a reference to the database; when s is found, p computes a noisy count of the number of rows in the database, which decreases the privacy budget.

The possibility of such attacks is acknowledged in the PINQ paper [20], and the PINQ implementation does contain hooks for an expression rewriter (called `Purify` in [20]) that is invoked on all user-supplied expressions and could potentially change or remove code that causes side-effects. However, such a rewriter is not provided; indeed, the PINQ downloads page contains an explicit warning that the code is not hardened or secured and should not be used 'in the wild.'

We conjecture that implementing a reliable `Purify` will be far from trivial. Avoiding the privacy budget attack will probably be easiest: every function that might consume privacy budget could be wrapped with a check that raises an exception if it is called from inside a running microquery (i.e., with a PINQ operation already on the call stack); this exception could then be turned into a default result for the microquery. State attacks are more difficult: since microqueries in PINQ are arbitrary bits of C#, it seems the choices are either to execute them on a modified virtual machine that detects writes to global state (as Airavat does), or else to create a small domain-specific language for writing microqueries that avoids global updates by design (as we do in Fuzz). Addressing timing attacks will require deeper changes to PINQ: the issues and available solutions are precisely the ones we study in this paper.

3.6 Case study: Airavat

Because Airavat calculates sensitivity and deducts the required amount from the privacy budget before query execution begins, it is inherently safe from privacy budget attacks. However, Airavat's mechanism for preventing state attacks permits a related vulnerability. To prevent microqueries from communicating via static variables, Airavat runs microqueries on a modified JVM; if a microquery ever attempts to modify a static variable, an exception is thrown and the whole query is marked "not differentially private." Unfortunately, the adversary can now observe whether the system gives them the result at the end of query execution or says, "Sorry, that's not differentially private." A better alternative would be to abort just the microquery, return, a default result, and allow the remainder of the query to run to completion.

In its published form, Airavat is also vulnerable to timing attacks. Its authors acknowledge this weakness [26] but counter that the bandwidth of the channel it creates is very low. This, we agree, may make it tolerable in some contexts, e.g., with "mostly trusted" queriers that might be careless but will not write malicious queries that intentionally attempt to reveal specific targeted secrets. We understand that Airavat may soon be enhanced to add timeouts to microquery executions [Shmatikov, personal communication, July 2010]; the implementation techniques described below should be useful in this effort.

4 Defending against timing attacks

State and privacy budget attacks can (and must) be addressed by designing the query language so that they are impossible. Timing attacks require more work, and this will be our concern for the remainder of the paper.

4.1 Four approaches to the problem

There are two basic strategies. One is to ensure that a given query takes very close to the *same* amount of time for *all possible* databases (of a given size—see below), so that the adversary can learn nothing from observing the time it takes the query result to arrive. The other is to treat time as an additional output of the query, and to limit the amount of information the adversary can gain using the same mechanisms (sensitivity analysis and ap-

propriate perturbation) that are used for data outputs.[2] In either approach, we can either obtain the information about running time statically (by analyzing the program before running it) or enforce limits dynamically (e.g., by using timeouts). This gives us the four possibilities shown in Table 1.

The solutions in the right-hand column provide somewhat weaker privacy guarantees than those on the left. In order to properly "noise" a resource like time, we must have the ability to both increase *and decrease* its consumption. While we can clearly increase execution time by adding a delay, we cannot easily decrease it. We can mitigate this problem by adding a default delay T; thus, we can add "time noise" $\nu \geq -T$ by delaying for $T + \nu$ at the end of each query. Nevertheless, since noise distributions guaranteeing differential privacy have unbounded support (i.e., $P(\nu) > 0$ for all ν), there is always a possibility that $\nu < -T$, in which case we cannot complete the computation. Thus, ε-differential privacy seems impossible in practice; all we can hope for is the slightly weaker property of (ε, δ)-differential privacy [11], where δ is a bound on the maximum *additive* (not multiplicative) difference between the probability of any given query output with and without a particular row in the input.

On the other hand, in the constant-time solutions (left column), the size of the database becomes public knowledge, since, except for the most trivial queries, execution time depends on the size of the database. In practice, this is probably a reasonable concession. In the case of the variable-time solutions (right column), the size of the database does not need to be published.

The static solutions (top row) are attractive in principle, but they depend on a static analysis of time sensitivity—something that has proved challenging except for very simple, inexpressive programming languages. We therefore concentrate on the bottom row. In this row, we choose one column to explore further: the "constant execution time" alternative, where we try to make each microquery take as close as possible to exactly the same amount of time. (The other column also deserves exploration; we believe similar mechanisms will be required.)

4.2 Default values

The approach we explore in the rest of this paper is to dynamically ensure that each microquery m takes the exact same amount of time T. If the microquery takes less time to execute, we delay it and only return its result after T. If the microquery has executed for time T without returning a result, we abort it. However, aborting the enclosing *macro*query is not an option because this would leak information to an adversarial querier. Instead, our approach is to have the microquery return a *default value* d in this case.

To avoid privacy leaks through the default value, d must not itself depend on the contents of the database. In Fuzz, a static value for d is included with the query. Also, for reasons that will become clear in Section 4.4, d should fall within the range of the microquery m.

4.3 Do default values decrease utility?

When the microquery for a row r times out while answering a non-adversarial query, the utility of the query's overall result almost inevitably degrades. After all, the result no longer incorporates the intended contribution of r or any other row whose microquery has timed out, but rather uses the default value for each such microquery. However, a non-adversarial querier can always avoid the inclusion of any default values by choosing a sufficiently high timeout. If the timeouts are chosen properly, *timeouts should never occur while answering non-adversarial queries*. Thus, the only querier who experiences degraded utility is the adversary.

The question, then, becomes how to choose the timeout values. One possible method is as follows. The querier is supplied with a reference implementation of the query processor that additionally outputs the maximum processing time T_{max} for each microquery. The querier can then (locally) test his queries on arbitrary databases of his own construction and thus infer a reasonable time bound. The querier then adds a small safety margin and uses, say, $1.1 \cdot T_{max}$ as the timeout for his query. He then submits the query to the actual query processor, to be run on the private database.

4.4 Do default values create privacy leaks?

At first glance, it may appear that default values are replacing one evil with another: they seem to plug the timing channel at the expense of introducing a data channel. However, this is not the case: as long as the timeouts are applied at the microquery level (as opposed to imposing a timeout on the whole query), differential privacy is preserved, for the following reason.

First, recall that Fuzz is designed to ensure that the completion time of a query depends only on the size of the database, but not its contents. Since we have assumed that the size of the database is public, and since our threat model rules out all the other channels, the only remaining way in which private information could 'leak' is through the (noised) data that the query returns.

Now, recall that the type system Fuzz implements is based on the type system from [25]. As described in [25], this type system ensures that all programs that type-check are differentially private. This is achieved by

[2] Note that the sensitivity analysis would have to account for interdependencies between a query's execution time and its output value, which is far from trivial.

inferring an upper bound on the program's sensitivity to small changes in its inputs—specifically, a change to an individual database row.

Fuzz extends the type system from [25] with microquery timeouts on `map` and `split`, but, crucially, timeouts do not increase the sensitivity of these two functions. The reason is that the sensitivity of `map` and `split` depends on the range of values that the microquery can return. Since the default value is taken from the range of values that the microquery can *already* return in the absence of timeouts, the addition of timeouts does not increase this range, and thus does not increase the sensitivity either.

Of course, running a query on a given database with and without timeouts (or with shorter vs. longer timeouts) can yield very different results. Suppose we have a database b and a function with microqueries that, without timeouts, produces an output o when it is run on b. If we now add a very short microquery timeout, we can easily cause *all* the microqueries to abort and return their default value, and the resulting output for the same database D can be dramatically different from o. However, this does not mean that differential privacy is violated. Recall from Section 2.1 that the differential privacy guarantee makes a statement about running *the same query* on two databases b and b' that differ in *exactly one row* r. If we run a query with timeouts on both b and b', the only microquery that could behave differently is the one on row r. All the other microqueries start in the same state for both databases, so their behavior will be exactly the same—they will either time out on both b and b', or on neither.

5 The Fuzz system

Next, we present the design of the Fuzz system, which represents one specific point (the lower left quadrant) in the solution space from Table 1. This point is a good first step because it works with existing programming-language technology and is relatively easy to implement.

5.1 Overview

Fuzz consists of three main components: a simple *programming language*, a *type checker*, and a *predictable query processor*. The programming language rules out channels based on global state or side effects, simply by not supporting any primitives that could produce either. The type checker rules out budget-based channels by statically checking queries before they are executed and rejecting any query that cannot be guaranteed to complete with the available balance. Finally, the predictable query processor closes timing-based channels by ensuring that each microquery terminates after very close to *exactly* a specified amount of time. Figure 4 illustrates our approach.

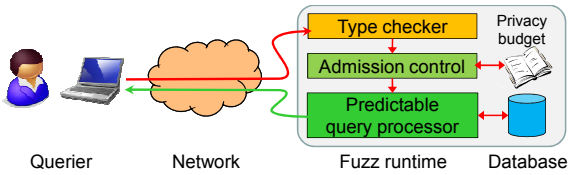

Figure 4: Scenario. Queries are first type-checked by Fuzz and then executed in predictable time.

5.2 Language and type system

Fuzz queries are written in a simple functional programming language whose functionality is roughly comparable to PINQ. The Fuzz language contains a special type `db` for databases, which is not a valid return type of any query. We say that a primitive is *critical* if it takes `db` as an argument. Our language ensures that critical primitives either return other values of type `db` (and nothing else) or add noise to all of their return values. Fuzz determines the correct amount of noise to add by using the sensitivity analysis and type system from [25].

Fuzz currently supports four critical primitives (Table 2): `map` applies a function f to each row in one database and returns the results in another database; `split` applies a boolean predicate p to each row in a database and returns two databases, one with all rows r for which $p(r) = \text{TRUE}$ and the other with the rest; `count` returns the (noised) number of rows in a database; and `sum` returns the (noised) sum of all the rows. `sum`'s type ensures that it can only be applied to databases with numeric rows.

5.3 Predictable query processor

To close timing channels, the query processor must ensure that all critical primitives take a predictable amount of time that depends only on the size of the database. This is trivial for `sum` and `count`. However, `map` and `split` involve arbitrary microqueries, and it can be difficult to statically analyze how much time these will take.

To avoid the need for such an analysis, Fuzz instead relies on *predictable transactions*. A predictable transaction is a primitive $\text{P-TRANS}(\lambda, a, T, d)$, where λ is a function, a an argument, T a timeout, and d a default value. P-TRANS takes *exactly* time T, and returns $\lambda(a)$ if λ terminates within time T, or d otherwise. Note that an implementation of P-TRANS may have to (a) add a delay if λ terminates early, and (b) abort λ slightly *before* T expires to ensure that any resources allocated by λ can be released in time. In Section 6, we describe two approaches to implementing P-TRANS in practice.

When evaluating `map` or `split`, Fuzz invokes P-TRANS for each microquery, using the specified timeout T and—in the case of `map`—the specified default value (`split` has an implicit default of TRUE).

Primitive	Arguments	Return value
`map db f T d`	Database `db`, function `f`, timeout `T`, default value `d`	Database
`split db p T`	Database `db`, boolean predicate `p`, timeout `T`	Two databases
`count db`	Database `db`	Noised $\|\texttt{db}\|$
`sum db`	Database `db`	Noised $\sum_i \texttt{db}_i$

Table 2: Critical primitives in the Fuzz language

All values of type `db` internally have representations of the same size, i.e., they consume the same amount of memory and (conceptually) have the same number of rows as the original database. If necessary, they are padded with dummy rows. For example, if the original database has 1,000 rows and consumes 1 MB of memory, the two databases returned by a `split` both consume 1 MB, and an invocation of `map` on either of them will invoke 1,000 microqueries—though of course the results of microqueries on dummy rows will be discarded.

5.4 How Fuzz protects privacy

We now briefly summarize how Fuzz protects against covert channels. First, the only observations a querier can make that depend on the contents of the database are the completion time of the query and its return value. This is because of (a) our threat model from Section 3.1, (b) the fact that the language contains no primitives with side-effects, such as mutating global state, and (c) the fact that the type system rules out abnormal termination.

Second, the return value of the query is differentially private. Since `db` is not a valid return type and critical operations return only values of type `db` or else appropriately noised values (based on the sensitivity that has been statically inferred [25]), the return value cannot depend on non-noised values from the database directly. Also, the language does not contain any primitives for observing side-effects within the query, such as memory consumption or the current wallclock time. The only time-related primitives are the timeouts on the microqueries; these have a sensitivity of 1 because (a) each microquery operates on only one row from the database at a time, and (b) microqueries have no access to global state and therefore cannot communicate with one another. Thus, if we add or remove one individual's data from the database, this affects only one row, so this can only cause one more (or less) microquery to time out and add a default result to the output.

Third, the completion time of a query depends only on the size of the database (which we assumed to be public) and data that has already been noised. To see why, consider that the only operations that have access to non-noised data are the microqueries, for which Fuzz enforces a constant runtime (by aborting or padding them to their timeout), and that values of type `db` cannot affect the control flow directly, only indirectly through return values of critical operations, which are noised. It is perfectly OK for the completion time of a query to depend on noised data, since such data is safe to release and could even have been returned to the querier directly.

In summary, Fuzz is designed to ensure that everything observable by the querier—whether directly through the data channel or indirectly through the timing channel—either does not depend on the contents of the database or has been noised appropriately.

6 Implementation strategies

In this section, we describe the abstract requirements for implementing predictable transactions, and we propose two concrete implementation strategies: one for newly designed runtimes (6.2) and one for retrofitting Fuzz into an existing runtime (6.3). Naturally, we expect the former to be more efficient and the latter to be easier to implement.

6.1 Requirements

To implement P-TRANS(λ, a, T, d), the following three properties need to hold for the language runtime:

- **Isolation:** $\lambda(a)$ can be executed without interfering with the succeeding computation in any way, apart from contributing its return value.
- **Preemptability:** The execution of $\lambda(a)$ can be aborted at any time, or at most within some time bound Δ_a;
- **Bounded deallocation:** At any point during the execution of $\lambda(a)$, there is a upper bound Δ_d on the time needed to deallocate all resources allocated so far by $\lambda(a)$.

If these requirements hold, we can implement P-TRANS by running $\lambda(a)$ in isolation and setting a timer to $T - \Delta_a - \Delta_d$ (which must be updated when Δ_d changes due to new allocations). If the timer fires, we can abort λ and deallocate its resources without overrunning the overall timeout T. After a final delay to reach T exactly, we can return either the result of $\lambda(a)$ if we have it, or d otherwise.

6.2 White-box approach

If we design a new language runtime from scratch, or if we are willing to make extensive changes to an existing runtime, we can achieve isolation and preemptability

by avoiding global variables that could be left in an inconsistent state when a microquery is aborted, as well as any termination of the microquery that does not correctly return the default value. Thus, it becomes possible to abort a microquery simply by performing a `longjmp` or its equivalent.

Regarding bounded deallocation, we expect that the key resource in most cases will be memory. It is possible to design the memory allocator in such a way that the memory allocated by a microquery can be deallocated in *constant* time. For example, we can divert the allocator from its usual allocation pool while a microquery is in progress, and instead allocate memory from a special region dedicated to microqueries. If the microquery takes arguments and returns results by value rather than by reference, objects in the main heap cannot acquire references to this region, so it is safe to summarily deallocate the entire region when the microquery aborts or terminates.

6.3 Black-box approach

The first strategy assumes a fairly deep understanding of how all primitive operations of the language are implemented, and how they interact with the allocator and each other. If we are working with an existing runtime system, it may be hard to be sure that the entire rest of the state of memory outside the microquery allocation region has been restored to its original state after a microquery finishes; for example, if we use any off-the-shelf library functions, they may have local buffers or other global state through which information can leak.

In this case, we can still ensure isolation and preemptability by leveraging operating system support, e.g., by farming out microqueries to a separate process, which can then be destroyed at any time without interfering with the state of the main runtime. Bounded deallocation can be achieved if we know an upper bound on the amount of time the operating system needs to destroy a process.

7 Proof-of-concept implementation

Next, we describe our proof-of-concept implementation of Fuzz. Our implementation does not execute Fuzz programs directly; rather, we implemented a front-end that accepts Fuzz programs, typechecks them, and then (if successful) translates them into Caml programs. Thus, we did not need to implement an entire language runtime from scratch; it was sufficient to implement a library with Fuzz-specific primitives like `map` and `split`, and to extend an existing runtime with support for predictable transactions. We chose Caml because it is similar enough to Fuzz to make the translation relatively straightforward.

7.1 Background: Caml Light

Our implementation is based on Caml Light [5, 19] version 0.75, a stable and lightweight implementation of Caml. Here, we briefly describe only the aspects of Caml Light that are relevant for our discussion of Fuzz. For a detailed description of Caml Light, please see [19].

In Caml Light, Caml code is first compiled into bytecode for an abstract machine called ZAM (the ZINC abstract machine); this bytecode is then executed on a runtime that implements the ZAM. Because of this architecture, the actual ZAM runtime is relatively simple: it mainly consists of an interpreter for the ZAM instructions and some code for I/O, memory management, and garbage collection.

The state of the ZAM consists of a code pointer, a register holding the current environment, an accumulator, two stacks (an argument stack and a return stack) and the heap. The heap is divided into two zones: a fixed-size 'young' zone and a variable-size 'old' zone. Most objects are initially allocated in the young zone; when this zone fills up, a 'minor' garbage collection copies any objects that remain active into the old zone. This was originally done to reduce the frequency of 'major' garbage collection runs (since most objects are short-lived, their space can be reclaimed very quickly), but it is also very convenient for Fuzz, as we shall see below.

Note that Fuzz uses the ZAM runtime to run only programs that it has previously translated from Fuzz programs. Thus, we can safely ignore features of the ZAM runtime (such as reference cells) that Fuzz does not use. Our threat model assumes that the adversary can submit only Fuzz programs, so he or she is unable to access any of these features.

7.2 Bounded deallocation

When a microquery times out, Fuzz must be able, within a bounded amount of time, to release all of the resources the microquery may have allocated. To this end, our implementation performs a minor collection at the beginning of each macroquery, which clears the young zone of the heap, and it confines any additional memory allocations during microqueries to the young zone. Thus, we can simply discard the *entire* young zone after each microquery, which requires only a single instruction. If the microquery completes normally (without a timeout), it writes its result into a special fixed-size buffer that is not part of young zone. If this buffer is empty after the microquery or contains only a partial result, the macroquery uses the default value instead.

Discarding the entire young zone is safe because, after a microquery, there cannot be any outside references to objects in that zone. Any new memory allocations must be in the young zone, any new values on the stacks are discarded as well, and the only objects in the old zone

that could be modified in place are reference cells, which translated Fuzz programs cannot use. Note that discarding the young zone is faster than a minor collection, so this particular modification (which is only possible for Fuzz programs, not for arbitrary Caml programs) actually results in a speedup.

7.3 Preemptability

Fuzz must be able to preempt a running microquery after a specified time, with high precision. To this end, our implementation creates a second thread that continuously spins on the CPU's timestamp counter (TSC).[3] When a microquery is started, the interpreter sets a shared variable to the time at which the preemption should occur; when that point is reached, the second thread sends a signal to the interpreter thread. To prevent the two threads from slowing each other down, each is pinned to a different CPU core. If the microquery terminates before the timeout, it simply spins until the preemption occurs.

Preemptions can occur at arbitrary points in the runtime code. To avoid inconsistencies, our implementation checkpoints all mutable state before each microquery; when the signal is raised, it uses `longjmp` to return to the macroquery and then restores the runtime state from the checkpoint. We exclude from the checkpoint any state that either is immutable or is discarded anyway – including both zones of the heap and any existing values on the stacks. This leaves just a handful of variables, such as the ZAM's stack pointers and the code pointer.

7.4 Isolation

Fuzz must ensure that a microquery cannot interfere with the rest of the computation in any way, other than contributing its return value. In the previous two sections, we have already seen that the states of the ZAM runtime before and after a microquery are logically equivalent, since any changes (other than the result value) are either discarded or rolled back. To avoid direct timing interference between microqueries, Fuzz also pads the runtime of the preemption code to $\Delta_a + \Delta_d$. However, Fuzz must also avoid indirect timing interference through the garbage collector, or from the rest of the system.

Fuzz prevents data-dependent invocations of the garbage collector by padding all database rows to consume the same amount of memory, and by padding all database objects to have the same number of rows. For databases that result from a `split`, Fuzz adds an appropriate number of dummy rows that consume memory and computation time but do not contribute to the result. Fuzz also disables the garbage collector during microqueries; if a microquery attempts to allocate more space than is available in the young zone of the heap, Fuzz stops it and forces it to time out. Thus, from the perspective of the macroquery (and the garbage collector), memory usage does not depend on un-noised values from the database.

To prevent page faults and context switches, Fuzz preallocates and pins all of its memory pages, and it assigns itself a real-time scheduling priority. In our experiments, this was sufficient to control the timing variations to within a a few microseconds.

[3] There are many other ways of implementing preemptions, such as periodic TSC checks in the interpreter loop, or using the CPU's performance counters.

7.5 Implementation effort

Altogether, we added or modified 6,256 lines of code, including 4,887 lines of C++ for the typechecker/translator, 1,119 lines of C++ and Caml code for our implementation of predictable transactions, 186 lines of C++ for benchmarking support, and 64 lines of Fuzz code for common library functions. For comparison, the entire Caml Light codebase consists of 29,984 lines of code. This supports our claim that Fuzz can be retrofitted into existing runtimes.

7.6 Limitations

Despite all our precautions, some potential sources of variability remain. For example, our current implementation does not freeze or flush the CPU's caches (since instructions like `wbinvd` are not available from user level), and it is designed to run on a commodity Linux kernel. We believe that these sources would be difficult to exploit because the adversary cannot control the memory layout or force the runtime to invoke system calls; also, any exploitable variation would have to be large enough to cause the $\Delta_a + \Delta_d$ padding to be overrun. An implementation with at least some kernel support could remove some or all of these sources, and thus use a less conservative padding.

8 Evaluation

Our evaluation has two primary goals. First, we need to demonstrate that Fuzz is *practical*, in the sense that it is sufficiently fast and expressive to process realistic queries. Second, we need to demonstrate that our Fuzz implementation is *effective*, i.e., that it prevents all the covert-channel attacks that are possible in our threat model (Section 3.1).

8.1 Non-adversarial queries

To demonstrate that Fuzz is powerful enough to support useful queries, we implemented three example queries that were motivated in prior work [4, 6, 12]. The **weblog** query is intended to run on the log of an Apache web server; it computes a histogram of the number of web requests that came from specific subnets. The **kmeans** query clusters a set of points and returns the three cluster

Name	Type	LoC	Inspired by
kmeans	Clustering	119	[4]
census	Aggregation	50	[6]
weblog	Histogram	45	[12]

Table 3: Examples of non-adversarial Fuzz queries.

centers, and the **census** query runs on census data and reports the income differential between men and women.

Table 3 reports the lines of code needed for each query. The queries are small because programmers only need to specify the actual data processing; parsing and I/O are handled by Fuzz. Also, the queries use a small library of generic primitives, such as lists and a `fold` operator, that consists of 64 lines written directly in the Fuzz language. Note that Fuzz can automatically certify queries as differentially private and perform sensitivity analysis during typechecking, so even non-experts can easily write differentially private queries.

8.2 Experimental setup

To evaluate the performance and effectiveness of Fuzz, we performed experiments using a setup consistent with our model from Section 3.1. We installed Fuzz on a dedicated machine, a Dell Optiplex 780 with a 3.06 Ghz Intel Core 2 Duo E7600 processor and 4 GB of memory. The machine was running a 32-bit Ubuntu Linux 11.04 with a 2.6.38-8 kernel. For our timing measurements, we used the CPU's timestamp counter, which is cycle-accurate. To minimize interference, we disabled CPU power management and the flush daemon, we kept all mutable data in a ramdisk and mounted all other file systems read-only, and we terminated all other processes on the machine, leaving Fuzz as the only running process (recall our assumption that the machine is dedicated to Fuzz). As discussed in Section 7.6, there are sources of timing variability that we could not disable, such as the periodic timer interrupt, which takes about 3 μs to handle in this setup, but these cannot be influenced by an adversary, so they merely add noise to the query completion time without leaking information. The padding time, which corresponds to $\Delta_a + \Delta_d$, was set to 10 μs; this setting was chosen to be the highest preemption latency we observed, plus a generous safety margin.

To estimate the overhead of our implementation, we also prepared a version of the three translated Fuzz queries that can run on the original Caml Light runtime. Since the original runtime does not support P-TRANS or a fixed-size memory representation for databases, this required small modifications to the Caml code; for example, the modified queries invoke microqueries without any timeouts, and they keep the database in ordinary Caml lists. These modifications do not affect the data output of the queries. We used the modified Caml code only for experiments with the original Caml Light runtime; all other experiments directly use the Caml code that is output by the Fuzz front-end.

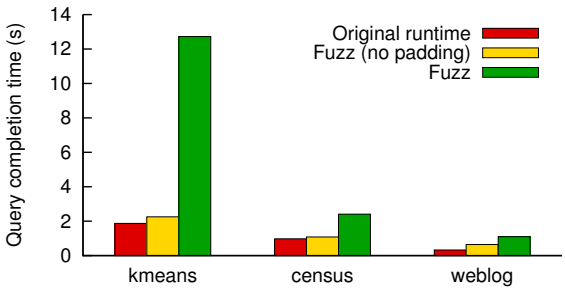

Figure 5: Performance for non-adversarial queries.

8.3 Macrobenchmarks

To estimate the performance of Fuzz, we ran each of the example queries from Table 3 over a synthetic dataset and measured the query completion time. Using synthetic data rather than real private data does not affect our measurements because, by design, the completion time does not depend on the contents of the database. However, the data *format* was based on realistic data—specifically, the weblog input was based on an Apache server log and the census input was based on U.S. census data from [14]. The synthetic database in each case had 10,000 rows. We set the microquery timeouts for each `map` and `split` by first running the query over example data with timeouts and padding disabled, measuring the maximum time taken by any of the `map` or `split`'s microqueries, and then setting the timeout to be 10% above that. We verified that no timeouts occurred during our measurements.

Figure 5 shows the query completion time for three different configurations: the original Caml Light runtime, the Fuzz runtime with both timeouts and padding disabled, and the Fuzz runtime with all features enabled. As expected, Fuzz takes more time to complete the queries than the original runtime; for our three queries, the slowdown was between 2.5x (census) and 6.8x (kmeans). However, in absolute terms, the completion times were not unreasonable: the most expensive query (kmeans) took 12.7s to complete, which seems low enough to be practical.

Figure 5 also shows that, with timeouts and padding disabled, Fuzz's performance is roughly comparable to that of the original Caml Light runtime. This is not an apples-to-apples comparison; for example, the fixed-size memory representation for databases costs performance, whereas erasing the young zone after each microquery is actually faster than garbage-collecting it. Nevertheless,

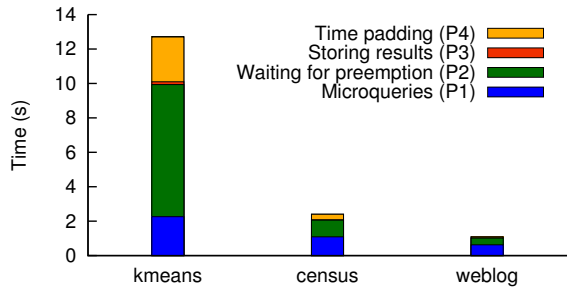

Figure 6: Time spent in different phases of query processing.

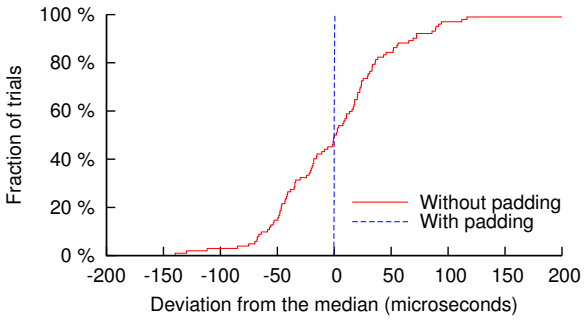

Figure 7: Variation of completion time for the weblog-delay query.

the numbers suggest that most of the overhead comes from padding and timeouts. Next, we examine this in more detail.

8.4 Microbenchmarks

To get a better picture of what factors influence the performance of our implementation, we added instrumentation in such a way that query time can be attributed to one of the following five phases:

- **P1:** Computation performed by a microquery;
- **P2:** Waiting for the preemption when a microquery completes early;
- **P3:** Preemption handling, storing results, restoring checkpoints, and loading the next row;
- **P4:** Padding the time of the preemption handler to $\Delta_a + \Delta_d$; and
- **P5:** Computation performed by the macroquery.

Figure 6 shows our results (we omit the time P5 taken by the macroquery because it was below 0.2% of the total for all queries). As already suggested by the previous section, the majority of the time is spent in either the waiting or the padding phase. This may seem rather conservative at first, but recall that the completion time of even a non-adversarial microquery can vary with the row it is processing; the timeout needs to be sufficient for the longest query with high probability. Timeout handling, deallocation, checkpointing, and storing the results takes comparatively little time.

Note that the overhead for the kmeans query is considerably higher than for the others. This is because kmeans repeatedly uses `split` to partition the database – specifically, to map each point to the nearest of the three cluster centers. Since our proof-of-concept implementation is not keeping track of the fact that the union of the three partitions contains exactly the N rows in the original database, it must conservatively assume that *each* partition might contain *all* the N rows. Thus, functions that operate on the partitions are padded to $3 \cdot N$ times the timeout, when in fact N times would be sufficient. This could be avoided by extending Fuzz with a suitable operator, e.g., a GroupBy as in PINQ.

8.5 Adversarial queries

As explained in Section 5.4, Fuzz rules out state attacks and privacy budget attacks by design, and it prevents timing attacks by enforcing that each microquery takes precisely the time specified by its timeout. This last point cannot be perfectly achieved by a practical implementation running on real hardware; we need to quantify how close our implementation comes to this goal.

To this end, we implemented five adversarial queries, exploiting different variants of the attacks from Section 3 to try to vary the completion time based on whether or not some specific individual is in the database:

- **weblog-delay** adds an artificial delay in each microquery that finds a match;
- **weblog-term** adds an artificial delay *except* when a microquery finds a match;
- **weblog-mem** consumes a lot of memory when a matching individual is found;
- **weblog-gc** creates a lot of garbage on the heap by repeatedly allocating and releasing memory;
- **census-delay** looks for a particular known person in the database and adds a timing delay if their income is above a specified threshold.

We ran each query on two versions of the corresponding database: one that contains the individual (Hit) and another that does not (Miss). To demonstrate the effectiveness of these attacks on an unprotected system, we first performed the experiment with Fuzz runtime and then repeated it with the original Caml Light runtime. This gives us four configurations per query. We ran 100 trials

Query	Attack type	Caml Light runtime (not protected)			Fuzz runtime (protected)		
		Hit	Miss	|Hit−Miss|	Hit	Miss	|Hit-Miss|
weblog-mem	Memory allocation	1.961 s	0.317 s	1.644 s	1.101 s	1.101 s	<1 μs
weblog-gc	Garbage creation	1.567 s	0.318 s	1.249 s	1.101 s	1.101 s	<1 μs
weblog-delay	Artificial delay	1.621 s	0.318 s	1.303 s	1.101 s	1.101 s	<1 μs
weblog-term	Early termination	26.378 s	26.384 s	0.006 s	1.101 s	1.101 s	<1 μs
census-delay	Artificial delay	2.168 s	0.897 s	1.271 s	2.404 s	2.404 s	<1 μs

Table 4: Effect of various attacks without and with predictable transactions. Each adversarial query tries to vary its completion time based on whether some specific individual is in the database. We show the total macroquery processing times when the individual is present (hit) and absent (miss), as well as the differences.

for each configuration, after a warm-up phase of two trials to ensure that the Fuzz binary and the database were in the file system caches.

Figure 7 shows how the completion times varied across the 100 trials, using the weblog-delay query with the Miss database as an example. With the original runtime, the completion times varied by approximately ± 150 μs around the median. With the Fuzz runtime, the completion times are extremely stable: the difference between maximum and minimum was <1 μs. The results for the other queries were similar, indicating that Fuzz's padding mechanism successfully masks internal variations between trials. Hence, we only report median values here.

Table 4 shows our results for the different configurations. We make the following three observations. First, the attacks are very effective when protections are disabled. For four out of the five queries, the completion times for the Hit cases were at least one second different from the completion times for the Miss cases, so an adversarial querier could easily have distinguished between the two cases and thus learned with certainty whether or not the individual was in the database. We could have achieved even higher differences simply by changing the queries. For weblog-term, the difference was only a few milliseconds; the reason is that, in order to change the completion time of the query by one second through early termination, the adversary would have had to make *each* microquery take at least one second, so the overall query would have taken a conspicuously long time – in this case, nearly three hours.

Second, the attacks cease to be effective in Fuzz. In each case, the difference between Hit and Miss is so small we could not even reliably measure it locally on the machine (for comparison, handling a timer interrupt requires about 3 μs, and one hundred of these are triggered every second, limiting the achievable accuracy), much less across a wide-area network, using the small number of trials that the privacy budget allows.

Third, the completion times are higher when protections are enabled. This is consistent with our earlier observations from Section 8.3.

8.6 Summary

Our results show that Fuzz is effective: it eliminates state and budget channels by design, and narrows the timing channel to a point where it ceases to be useful to an adversary. Query completion times remain practical but are substantially higher than in an unprotected system.

9 Related Work

Differential privacy: There is a considerable body of work on the theory of differential privacy [8–10] and on differentially private data analysis [20, 26]. Except for the papers on Airavat [26] and PINQ [20], none of these papers discuss covert-channel attacks by adversarial queriers. The PINQ paper briefly mentions certain security issues, such as exceptions and non-termination; Airavat discusses timing channels, but, as we have shown in Section 3.5, its defense is not fully effective. The present paper complements existing work by providing a practical defense against covert-channel attacks, which could be applied to existing systems.

Covert channels: Covert channels have plagued systems for decades [18, 30], and they are notoriously hard to avoid in general. Fuzz is a domain-specific solution; it only addresses differentially private query processing, but it can give strong assurances in this specific setting.

A variety of defenses against covert channels have been suggested. Most related to this paper is the work on external timing channels. The bandwidth of external timing channels can be reduced, e.g., by adding random delays [15, 16] or by time quantization [2]. However, to guarantee differential privacy, the adversary must be prevented from learning even a *single* bit of private information with certainty, so a mere reduction in bandwidth is not sufficient in our setting. Fuzz avoids this problem by converting the timing channel into a storage channel, which in turn is handled by differential privacy.

Preventing timing channels seems hopeless in the general case. Language-based designs can eliminate them for certain types of programs [1], but only at the expense of severely limiting the expressiveness of the programming language. Shroff and Smith [27] show how to handle more general computations but may have to abort

them, which can result in garbled data and/or leak information through a storage channel. In the context of a differentially private query, however, aborting individual microqueries is safe because the impact on the overall result is known to be bounded by the sensitivity of the query. As shown in Section 4.4, returning default values does not open a new storage channel or increase the privacy cost of the query (though it may decrease its usefulness).

Side channels: Side channels can leak private information, e.g., through electromagnetic radiation [13, 24] or power consumption [17]. Many of these channels can only be exploited if the adversary is physically close to the machine that executes the queries, which is not permitted by our threat model.

Real-time systems: Some real-time systems have provisions for handling timer overrun problems in untrusted code, such as preemption or partial admission [29]. In our scenario, it would not be sufficient to simply preempt a microquery that has overshot its timeout—we must be able to terminate it *and* clean up all of its side effects *before* the timeout expires. Another approach is inferring the worst-case execution time [28], which is known to be difficult even for trusted code.

10 Conclusion

We have demonstrated that state-of-the-art systems for differentially private data analysis are vulnerable to several different kinds of covert-channel attacks from adversarial queriers. Covert channels are particularly dangerous in this context because the leakage of even a single bit of private, un-noised information completely destroys the guarantees these systems are designed to provide. We analyzed the space of potential solutions, and we presented the design of Fuzz, which represents one specific solution from this space and relies on default values and predictable transactions. Using a proof-of-concept implementation based on Caml Light, we demonstrated that Fuzz can be retrofitted into an existing language runtime. Our evaluation shows that Fuzz is practical and expressive enough to support realistic queries. Fuzz increases query completion times compared to systems without covert-channel defenses, but the increase does not seem large enough to prevent practical applications.

Acknowledgments

We thank Jason Reed for his contributions to the early stages of this project, and Frank McSherry, Vitaly Shmatikov, Trent Jaeger, Helen Anderson, our shepherd Miguel Castro, and the anonymous reviewers for their helpful comments. This research was supported in part by ONR Grant N00014-09-1-0770 and by US National Science Foundation grants CNS-1065060 and CNS-1054229.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM POPL*, Jan. 2000.

[2] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. ACM CCS*, Oct. 2010.

[3] M. Barbaro and T. Zeller. A face is exposed for AOL searcher No. 4417749. *The New York Times*, Aug. 2006. `http://select.nytimes.com/gst/abstract.html?res=F10612FC345B0C7A8CDDA10894DE404482`.

[4] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proc. PODS*, June 2005.

[5] Caml Light website. `http://caml.inria.fr/caml-light/index.en.html`.

[6] S. Chawla, C. Dwork, F. McSherry, A. Smith, and H. Wee. Toward privacy in public databases. In *Proc. TCC*, Feb. 2005.

[7] S. Crosby, D. Wallach, and R. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security*, 12(3):1–29, 2009.

[8] C. Dwork. Differential privacy. In *Proc. ICALP*, July 2006.

[9] C. Dwork. Differential privacy: A survey of results. In *Proc. 5th Intl Conf. on Theory and Applic. of Models of Comp.*, 2008.

[10] C. Dwork. The differential privacy frontier (extended abstract). In *Proc. IACR TCC*, Mar. 2009.

[11] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. EUROCRYPT*, May 2006.

[12] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.

[13] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Proc. CHES*, May 2001.

[14] S. Hettich and S. D. Bay. The UCI KDD archive. Univ. of California Irvine, Dept. of Information and Computer Science, `http://kdd.ics.uci.edu/`.

[15] W.-M. Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, May 1991.

[16] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network pump. *IEEE Trans. Softw. Eng.*, 22:329–338, May 1996.

[17] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. CRYPTO*, 1999.

[18] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, Oct. 1973.

[19] X. Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[20] F. McSherry. Privacy integrated queries. In *Proc. ACM SIGMOD*, June 2009.

[21] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the net. In *Proc. ACM KDD*, 2009.

[22] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE S&P*, May 2008.

[23] PINQ website. `http://research.microsoft.com/en-us/projects/pinq/`.

[24] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Proc. Intl. Conf. on Research in Smart Cards (E-SMART)*, Sept. 2001.

[25] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, Sept. 2010.

[26] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, 2010.

[27] P. Shroff and S. F. Smith. Securing timing channels at runtime. Technical report, The Johns Hopkins University, July 2008.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[29] M. Wilson, R. Cytron, and J. Turner. Partial program admission. In *Proc. IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, Apr. 2009.

[30] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, May 1991.