McFIL: Model Counting Functionality-Inherent Leakage

Maximilian Zinkus, Yinzhi Cao, and Matthew D. Green, *Johns Hopkins University*

https://www.usenix.org/conference/usenixsecurity23/presentation/zinkus

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

McFIL: Model Counting Functionality-Inherent Leakage

Maximilian Zinkus
Johns Hopkins University
zinkus@cs.jhu.edu

Yinzhi Cao
Johns Hopkins University
yzcao@cs.jhu.edu

Matthew D. Green
Johns Hopkins University
mgreen@cs.jhu.edu

Abstract

Protecting the confidentiality of private data and using it for useful collaboration have long been at odds. Modern cryptography is bridging this gap through rapid growth in secure protocols such as multi-party computation, fully-homomorphic encryption, and zero-knowledge proofs. However, even with provable indistinguishability or zero-knowledgeness, confidentiality loss from leakage *inherent to the functionality* may partially or even completely compromise secret values without ever falsifying proofs of security.

In this work, we describe McFIL, an algorithmic approach and accompanying software implementation which automatically quantifies intrinsic leakage for a given functionality. Extending and generalizing the Chosen-Ciphertext attack framework of Beck *et al.* with a practical heuristic, our approach not only quantifies but *maximizes* functionality-inherent leakage using Maximum Model Counting within a SAT solver. As a result, McFIL automatically derives approximately-optimal adversary inputs that, when used in secure protocols, maximize information leakage of private values.

1 Introduction

Functionality-inherent leakage (FIL) is a universal characteristic of systems which compute over private data. Modern cryptography has enabled many such systems, including secure multiparty computation (MPC), fully-homomorphic encryption (FHE) [18], and zero-knowledge (ZK) proofs [23]. The use of these cryptographic tools is growing steadily across the public and private domains, increasing the stakes of these systems' security and increasingly bifurcating the set of people who develop these systems from the set who use and rely on them.

FIL occurs when an adversary is able to observe or participate in computation over private data, and observe the outputs of this computation (or factors correlated with them). Naturally, given a computable function and even partial knowledge of some inputs and outputs, an adversary can infer *some*thing about the unknown inputs which induced observed outputs.

In this work, we provide McFIL, an algorithmic approach for evaluating FIL within arbitrary functionalities, and an accompanying software implementation which can be used to determine the extent of leakage a functionality admits. Our approach does not exploit any cryptographic insecurity; rather, we leverage the unavoidable leakage inherent to underlying functions. The fact that cryptographic protocols can reveal information via correctly-evaluated outputs should be no surprise to cryptographers. It is our aim to provide a systematic way for prospective, non-expert users of cryptographic systems to evaluate functionalities they wish to compute securely, so they can make informed decisions on the risks of doing so even within cryptographically-secure schemes.

Our approach. We rely on a family of techniques called Model Counting, and we refer to our tool as McFIL for "Model Counting Functionality-Inherent Leakage." Most critically, McFIL is designed to quantify and optimize leakage *given only a description of the circuit to be implemented*, and does not require the implementer to assist the tool in understanding the functionality. This allows the tool to automatically derive a number of "attacks," including those not easily predicted by practitioners. For example:

- The classic Yao's Millionaires problem [47] admits one bit of leakage per execution due to the functionality (greater-than comparison). Given only a description of the functionality, McFIL automatically derives inputs to uncover the other player's salary in approximately $log(n)$ sequential executions (Figure 1).
- Dual Execution MPC [22, 30] admits adversary-chosen one-bit leakage in an equality check protocol. McFIL derives a sequence of predicates of configurable complexity to uncover the honest party's input.
- We evaluate McFIL against an array of other functionalities as proofs of concept in §5, either completely recovering the honest parties input(s) or providing an equivalence class of candidate solutions many orders of magnitude smaller than the initial search space.

Leakage Explained. When considering novel functionalities for use in secure protocols such as MPC, FHE, or ZK, one must consider how the privacy of secret inputs will be maintained. Therefore, practitioners must consider the security of their schema but also what can be inferred from the outputs of the functionalities. Even with provably secure protocols, the confidentiality loss inherent to a given functionality may partially or even completely leak secret values without ever violating the security guarantees of a protocol.

While functionality-based leakage may be unavoidable, it is quantifiable. Prior works [13, 27–29] have applied various information flow and optimization techniques to this problem. However, quantifying information flow [12] over programs is NP-hard and grows in the complexity of the program state space [25]. As a result, these approaches have remained computationally infeasible in practice. In addition to these general results, bespoke analyses for individual protocols or leakage paradigms exist in the literature [1, 6, 20], but these require manual analysis effort by experts which does not scale.

Contributions. We provide a practical methodology for *automatically quantifying* and even *optimizing over* inherent leakage of a circuit-based functionality. Our tool McFIL can be used to analyze privacy in MPC, FHE, or ZK. We generalize and extend a series of techniques developed by Beck *et al.* [4] in automating chosen-ciphertext attacks (CCA) in order to bring their work to a far broader class of functionalities.

In summary, our contributions are as follows:

- We describe McFIL, an algorithmic approach to quantifying and exploiting information leakage for a given functionality. (§4)
- We generalize and extend the Delphinium cryptanalysis framework of Beck *et a.* [4] beyond its original domain of strictly-defined predicate functions, addressing key limitations of that work and resolving an open question from it. (§4.1)
- We provide a software implementation [49], as an artifact accompanying this submission and open-source tool. Our implementation encodes nontrivial domain knowledge to directly manipulate SAT instances and enable extensive parallelism. (§4.2)
- Finally, we provide SAT instances generated from McFIL and our sample programs. The SAT community gathers such instances as benchmarks [21] to guide future SAT solver development, which will in turn expand the range and complexity of functions our tool can analyze within a given amount of computation time. (§5.3)

2 Intuition: Maximizing Leakage

In the following two illustrative examples, we analyze simple functionalities to manually derive the results which McFIL automates.

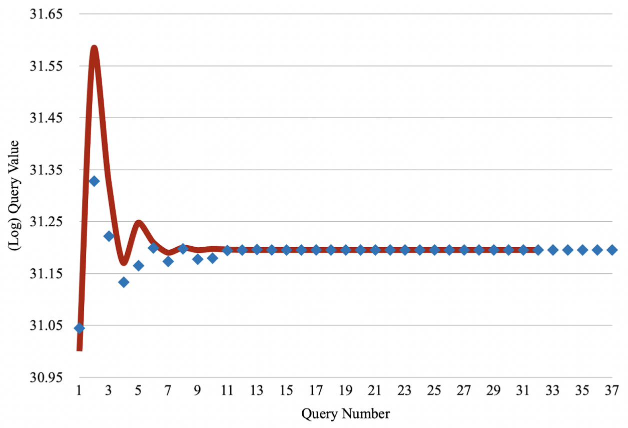


Figure 1: In a fully-automated analysis of Yao's Millionaires, McFIL "discovers" binary search. Blue diamonds are adversary inputs, true binary search shown in solid red.

2.1 A Minimal Example: Binary *AND*

Consider a simplistic two-party MPC (2PC) protocol which securely computes the functionality $\mathcal{F}$ consisting only of a Boolean *AND* gate ($\wedge$).

$$\begin{aligned}
&\mathbb{Z}_2 = \{0,1\}\\
&\mathcal{F}: \mathbb{Z}_2 \times \mathbb{Z}_2 \to \mathbb{Z}_2 = \wedge\\
&a \in \mathbb{Z}_2\text{: honest party's input}\\
&b \in \mathbb{Z}_2\text{: adversary's input}\\
&x \in \mathbb{Z}_2\text{: output } (x = a \wedge b)
\end{aligned}$$

Maximizing Leakage. After normal execution, the adversary should have no knowledge of a. However, knowing $\mathcal{F} = \wedge$, the adversary may *a priori* choose their input $b = 1$ to gain complete knowledge of a upon learning x:

$$\begin{aligned}
x = 0 &\implies a \wedge 1 = 0 \implies a = 0\\
x = 1 &\implies a \wedge 1 = 1 \implies a = 1
\end{aligned}$$

The choice of $b = 1$ *maximizes the information* gained from x. Had they chosen $b = 0$, $x = 1$ would have become impossible for the *AND* functionality, and thus the output would provide no information about a. By choosing b optimally, the adversary learns a despite the security of the MPC.

2.2 A Further Example: Yao's Millionaires

Now, consider the classic 2PC example of the Millionaires problem [47, 48]. Two parties (A, B) wish to compare their wealth (say, as 32-bit integers a and b) without disclosing anything aside from the predicate result of $a < b$.

$$\begin{aligned}
&\mathbb{Z}_{32} = \{0,\ldots,2^{32}-1\}\\
&\mathcal{F}: \mathbb{Z}_{32} \times \mathbb{Z}_{32} \to \mathbb{Z}_2 \;=\; <\\
&a \in \mathbb{Z}_{32}\text{: honest party's input}\\
&b \in \mathbb{Z}_{32}\text{: adversary's input}\\
&x \in \mathbb{Z}_2\text{: output } (x = a < b)
\end{aligned}$$

Maximizing Leakage. In this example, no single adversary input will uniquely constrain the honest party's input (at least, not in expectation over a uniform distribution of possible a).

Now, consider $b = 2^{31} - 1$. If $x = 0$ (`false`), the most significant bit of the honest input must be 1 as $a \geq 2^{31}$. Equivalently, if $x = 1$, then $MSB(a) = 0$. Assuming uniform a, in expectation the adversarial input $b = 2^{31} - 1$ eliminates half of the candidate solutions for a, providing the adversary a single bit of information.

Extending Leakage. On its own, this single bit of information out of 32 bits is potentially unimportant. The number of possible solutions for a remains large, and the adversary has no way to distinguish between them.

Intuitively, we next consider: *what if the adversary can try again?* Although this extends beyond the original threat model of many secure protocols, it does so with pragmatism: one can imagine settings where a protocol may be executed multiple times e.g. in a client-server model, when attempt-limiting protections are missing or evaded, or when an honest party may unwittingly interact with multiple colluding adversaries.

Maximizing *Multi-Run Adaptive* Leakage. Given additional attempts (or "queries," following [4]), a binary search emerges from the Millionaires comparison functionality. When the protocol may be repeated, a can be completely uncovered in $log(|\mathbb{Z}_{32}|) = 32$ queries, violating confidentiality despite a secure protocol, and making *optimally-efficient use* of the newly-assumed threat model of multiple queries.

This should be unsurprising to cryptographers: if security is proven modulo leakage, and leakage is allowed to grow, naturally security may be compromised. However, automatically analyzing and quantifying leakage over many queries is useful in two ways. First, the more intuitive: if a protocol does admit some method to retry, for example a smart contract [46] which may be repeatedly executed or an online oracle which does not sufficiently authenticate users, a multi-run leakage amplification attack may be possible. In this case, McFIL can be used to derive a set of queries to efficiently exploit leakage. Second, the multi-run setting can be useful in analyzing protocols which do not admit multiple attempts in that McFIL can compute an average leakage statistic per query. For arbitrary protocols, the true extent of leakage may be unknown; McFIL provides a way for practitioners to evaluate functionalities before choosing to encode them into secure protocols. As demonstrated in Figure 1, McFIL approximately rediscovers this binary search attack to uniquely identify the `target` secret input.

2.3 Overview of McFIL

Here we provide intuition as to how McFIL uses SAT solving to achieve automated leakage maximization. Refer to Figure 2 for a visualization of our automated workflow.

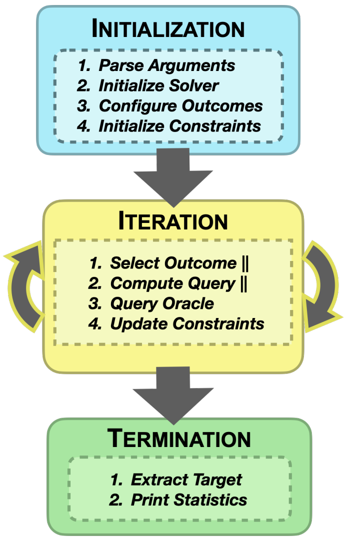


Figure 2: McFIL workflow; || denotes parallelization

Prior Work: Delphinium. Beck *et al.* applied SAT solving to the problem of Chosen Ciphertext attacks (CCAs) in their tool, Delphinium [4]. Their methodology formulates CCAs as a sequence of optimization problems over the outputs of a format oracle, automating attacks against classic and novel oracles to exploit CCA vulnerabilities.

Format oracles are predicate functions which determine if an input is well-formed or not, and Delphinium exploits this leakage to decrypt. In this work, we follow the methodology of Beck *et al.* and formulate generalized functional leakage as a similar sequence of optimization problems. The main contributions of this work, then, are the necessary adaptations to generalize their approach from predicates (1-bit outputs) to arbitrary n-bit output functions while maintaining concrete efficiency in real-world target functionalities. Further, Delphinium requires its format functions to be well-defined (e.g. for all possible inputs, either $true$ or $false$ is returned). To accurately capture the breadth of arbitrary functions, we release this requirement using a novel, concretely-efficient heuristic described in §4.1.

Initialization. To initialize McFIL, a Boolean circuit representation of a function functionality must be provided. Compilers exist to facilitate this process [16], and McFIL offers a Python3 DSL to facilitate encoding (refer to Appendix A). These instances contain symbolic bits representing the "target" secret input(s) to be uncovered (`target`), and symbolic bits representing the adversary's "chosen" input(s) (generated by McFIL) or "queries" (`chosen`).

Iterative Solution Elimination. In order to uncover secret input(s) in an automated, multi-round analysis, McFIL requires access to a concrete instantiation of the functionality. Whether implemented as a test shim for local execution (as we provide in our implementation) or e.g. a live instance of a secure protocol, this instantiation must compute the functionality given the adversary's query and the true secret input(s) of the honest parties, and provide the result back to McFIL. We refer to this as the *function oracle*, and each iteration corresponds to e.g. a single MPC evaluation in the *multi-run adaptive* threat model.

Using a SAT solver, these two components alone are sufficient to iteratively constrain the search space of the secret input(s). However, the queries generated at each iteration would be arbitrarily drawn from the set of satisfying models within the solver – degenerating to a brute-force guessing attack within an exponential search space. What remains is to optimize the *profitability* (expected number of eliminated models [4]) of each query.

Optimizing Queries. A truly optimal leakage maximization would consider all possible numbers and contents of queries. However, due to the underlying complexity of the problem, this approach fails in practice. Existing leakage analysis work in side channels [31] attempts to analyze a sliding window of multiple query-like values, and reaches computational limits in 8-bit secret domains. Therefore, we consider only maximizing at each iteration: a greedy-optimal approach.

In order to maximize profitability at each iteration, we employ Max#SAT to *simultaneously maximize* the number of satisfying solutions for `target` corresponding to all possible outputs of the function under test. McFIL then iteratively derives function inputs which imply partitions of the solution space, and eliminates one of the subsets of each partition based on the result from the function oracle.

Max#SAT, described in detail in §3, is a counting-maximization analogue to Boolean Satisfiability (SAT) which can be efficiently, probabilistically approximated [17]. Crucially, these approximation algorithms can themselves be efficiently encoded into SAT constraints [4]. By using these approximations as constraints in SAT instances, McFIL probabilistically and approximately ensures that a large number of satisfying solutions exist for given symbolic variables.

Extracting a Model. With all constraints in place and an unknown `target`, a satisfying solution (or "model") for `chosen` is extracted from the solver. The maximization constraints ensure that `chosen` is selected approximately optimally: for each possible output of $\mathcal{F}(\texttt{chosen}, \texttt{target})$, many candidate models for `target` exist.

Given only a symbolic representation of `target`, $\mathcal{F}$ cannot be used to compute a concrete result. However, when the concrete protocol instantiation – containing knowledge of the true value of `target` – is executed with the adversary's `chosen` input, a single outcome `result` is returned. Then, all candidate solutions for which $\mathcal{F}(\texttt{chosen}, \texttt{target}) \neq \texttt{result}$ may be eliminated. Critically, due to the maximization constraints, this set of eliminated candidates will be large.

Eliminating Candidates. After each query, all candidate models for `target` *inconsistent* with each $(\texttt{chosen}, \texttt{result})$ pair are eliminated. That is, they are no longer satisfying solutions for `target` in the evolved constraint system. This leads to a greedy algorithm in expectation, iteratively reducing the `target` search space in maximized increments.

Beyond Predicates. Problematically, the Delphinium algorithm is restricted to predicate functions, and intrinsically requires that for all `chosen`, every possible output for $\mathcal{F}$ is reachable. We refer to this informally as *completeness* in §4:

$$\forall \texttt{chosen}, \texttt{result}\ \exists \texttt{target}$$
$$s.t.\ \mathcal{F}(\texttt{chosen}, \texttt{target}) = \texttt{result}$$

This completeness restriction is often trivial for predicates. While this is sufficient for chosen-ciphertext attack discovery [4], as the relevant padding/format functions are generally predicates which determine message validity, arbitrary functionalities are not necessarily so amenable. Further, Delphinium requires the *operator* to carefully define Boolean formulae to adhere to this notion of completeness, requiring additional effort and expert insight. We describe this challenge and our approach to generalize and extend Delphinium in §4.1.

Negative Results: Demonstrating Security. Depending on the functionality, it may be impossible to differentiate any classes of candidates; McFIL detects this and provides a message that the attack may proceed as "brute-force." This negative result can also be taken as an indication that leakage is bounded for the given functionality. If little enough of the private input is derived before brute-force is required, this may be taken as a probabilistic argument for the security of the functionality against leakage-based attacks.

3 Technical Background

In this section, we discuss SAT, its extensions, and their relative complexity. We also highlight software tools solving or approximating these problems which have emerged from the SAT research community, and describe their relevance to McFIL. Finally, we review the limited past works which have broached function-inherent leakage and faced computational-feasibility limitations.

3.1 SAT and SMT

Boolean Satisfiability. SAT is a widely-known NP-complete problem. SAT takes as input a Boolean formula which relates a set of binary input values through Boolean operations. A solution to SAT is an assignment (also called a model, solution, witness, or mapping) of `true` and `false` (eq. 1 and 0) values to the Boolean inputs which causes the formula to evaluate to `true`. SAT formulae are commonly organized to include only *And*, *Or*, and *Not* operations in Conjunctive Normal Form (CNF) [42]. Any propositional formula can be efficiently converted into CNF preserving satisfiability, although conversion may introduce a linear increase in formula size [42]. In CNF, *literals* represent binary input values, and may be negated (denoted with $\neg$). These literals (negated or otherwise) are grouped into disjunctions (*Or*), which are themselves grouped into a single conjunction (*And*). In short, CNF is an "and of ors."

SAT Solvers. SAT solvers are tools designed to determine the (un)satisfiability of Boolean formulae. SAT solvers often require CNF, and provide a *model* in the event the formula is satisfiable. The model is not guaranteed to be unique (and often is not).

Due to the generality of NP-complete problems [14], SAT solvers are powerful tools. Since 1962, the DPLL [15] method of backtracking search has served as the core tool in SAT solving, with the more recent development of Conflict-Driven Clause Learning (CDCL) [37] in 1996 aiding in optimizing the search for a satisfying model or contradiction indicating UNSAT.

Satisfiability Modulo Theories. Given a SAT solver, the task remains to translate problems of interest into Boolean formulae. In order to aid translation, SMT solvers were developed. SMT solvers add expressive domains of constraint programming to SAT such as arithmetic and bitvector (bit-wise operations beyond single Boolean values) logic. SMT solvers such as Z3 [16] have enabled solving problems ranging from program verification [9, 19, 36] to type inference and modeling, and more.

3.2 Extensions to SAT

Despite the expressive power of SMT solvers, program analysis alone is insufficient for McFIL as we derive optimizations over the target circuit. As we *maximize* over a solution space of program results, McFIL generates instances of Max#SAT (described in this section) from the Boolean formula describing the functionality under test. Intuitively, Max#SAT is a strictly more complex problem than SAT, but can be approximated using the recent technique of Fremont *et al.* [17] and a SAT solver. In this section we provide technical background describing SAT through Max#SAT.

Definition 1: $Max\#SAT(\phi, \overline{X}, \overline{Y}, \overline{Z}) = \overline{X}_{max}, max$

Input: ϕ: Boolean formula;
$\overline{X}, \overline{Y}, \overline{Z}$: Vectors of Boolean inputs to ϕ

Output: $\overline{X}_{max}$: Assignment (concrete Boolean values) of variables in $\overline{X}$ for which the number of satisfying solutions to variables in $\overline{Y}$ is maximized and at least one satisfying solution exists in $\overline{Z}$;
max: Number of satisfying solutions for $\overline{Y}$ ($0 \leq max \leq 2^{|\overline{Y}|}$)

SAT to #SAT. #SAT, pronounced "sharp SAT," is the counting analogue to SAT. Also referred to as *model counting*, #SAT asks not only *if* a satisfying model exists, but *how many* such models exist. The result lies in a range from 0 for an unsatisfiable formula, to 2^n for a (completely unconstrained) formula over n bits. #SAT is #P-complete, at least as difficult as the corresponding NP problem, but this phrasing belies its complexity: Toda demonstrated PH $\subseteq$ P$^{\#P}$ [41], that a polynomial-time algorithm able to make a single query to a #P oracle can solve any problem in PH, the entire polynomial hierarchy (which contains both NP and co-NP) [40].

#SAT to Max#SAT. Max#SAT (denoted in Definition 1) is the optimization analogue to the #SAT counting problem. As defined by Fremont *et al.*, Max#SAT takes a Boolean formula, denoted ϕ, over Boolean variables divided into three notional subsets $\overline{X}, \overline{Y}, \overline{Z}$. A solution to Max#SAT provides a model for the variables in $\overline{X}$ such that the count (number of models) of the variables in $\overline{Y}$ is maximized and at least one model exists for the variables in $\overline{Z}$. ($\overline{Z}$ exists to allow variables to remain in the SAT instance, yet outside the optimization constraints). A Max#SAT solution is particularly useful when $\overline{X}$ refers to input variables to the formula, meaning that a model for $\overline{X}$ could be provided as an input to a program represented by the circuit ϕ. Correspondingly, $\overline{Y}$ should be configured to contain variables of interest for maximization such as those representing quantitative information leakage, probabilistic inferences, or program-synthesis values [17].

Approximating Max#SAT. Max#SAT is at least as complex as #SAT [17, 41], but as Fremont *et al.* demonstrates, this does not prevent approximating Max#SAT using a number of calls to an NP oracle – or in concrete terms, a SAT solver. To approximate Max#SAT, Fremont *et al.* rely on a sampling technique first described by Valiant [44], and expanded upon (and implemented in software) by Chakraborty and Meel *et al.* [11] and Soos [38]. By applying *almost-uniform* hash functions [38, 43], which representatively sample a domain with error, a non-uniform search space can be proportionately sampled with bounded error to enable a more feasible count. Crucially, these hashes can be efficiently sampled and applied within a SAT solver [10]. This approach is central to McFIL: using *almost-uniform* hash functions as constraints within the formula, large classes of models for `target` can be identified and iteratively eliminated.

3.3 Prior Work

Functionality-based leakage is a known problem, although practical quantification methods are lacking in the literature. Clarkson *et al.* [13] analyze adversary knowledge of private MPC inputs using quantified information flow, but provide only a theoretical treatment of the problem. This idea was later explored by Mardziel *et al.* [27–29] in their attempt to maximize adversary knowledge through probabilistic polyhedral optimization. Although their solution is theoretically robust, Mardziel *et al.* note the "prohibitive" computational cost of their approach in practice [27].

SAT solving has also been applied to MPC in the recent literature, however, with particular focus on the *intermediate values* generated during protocol execution. These intermediate values may leak some degree of information, which has been quantified using information flow [33] and language-based formalization methods [2]. Further, when intermediate values can be determined to leak only *negligible* information, prior work has shown they may be computed "in-the-clear" as an optimization [33].

Quantitative information flow, program analysis, and other automated approaches have also been applied in the detection and mitigation of side-channel leakage vulnerabilities. These vulnerabilities are similar to function-inherent leakage in that they provide additional signal which may compromise private inputs based on the (otherwise secure) execution of a function. A recent systematization [7] of side-channel detection literature enumerates various techniques ranging from micro-architectural modeling or even reverse engineering to program-analysis-like tools to analyze hardware descriptor languages for potential side channels. One such work, SLEAK [45], computes a statistical distance between secret values and active intermediate values in the processes using a full-system simulator. The information-theoretic metric of leakage they compute is likely correlated with the leakage which McFIL is able to uncover and maximize, however, their approach requires hardware simulation and a compiled binary, neither of which may be available or even relevant to target functionalities intended to be executed e.g. within an MPC.

4 McFIL

In this section, we describe our primary contribution, McFIL. We introduce the novel `SelectOutcomes` prioritization subroutine which extends the Max#SAT-based approach beyond *complete* predicate functions and enables its use in the domain secure protocols such as MPC, FHE, and ZK.

Attack Model. An ideal approach to identifying leakage would allow for the quantification of useful leakage after one round of execution. For novel function circuits, this analysis can inform operational security requirements and privacy considerations. However, in many cases a functionality will be executed multiple times on identical (or related) inputs.[1] Ensuring the safety of all private inputs in these cases can also be seen as a form of defense-in-depth. For this setting, a *multi-run adaptive* model allows McFIL to extract more information about the privacy implications of a given Boolean circuit, whether the goal is to evaluate the safety of allowing untrusted

[1]This can occur in some protocols by design. Alternatively it may occur through deception, corruption of honest parties, or a failure of access control.

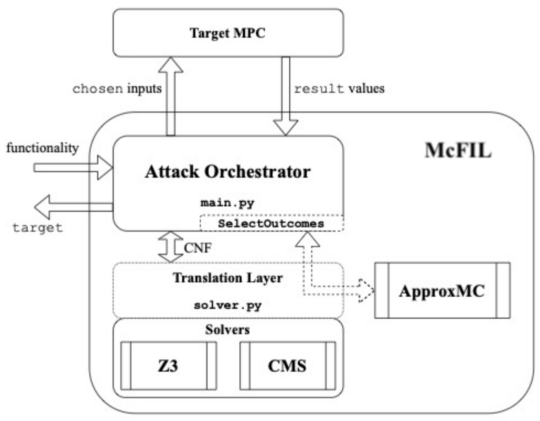


Figure 3: Architecture of McFIL

parties to execute the protocol repeatedly (to automate an attack), or simply to to quantify and bound overall leakage.

System Architecture. Figure 3 depicts the architecture of McFIL. Our tool integrates a custom CNF manipulation toolkit (`solver.py`) which efficiently represents constraints and orchestrates parallel solving instances.

4.1 Beyond Predicate Functions

In Delphinium [4], a format oracle classifies inputs as either valid or invalid according to a format specification and a predetermined fixed input length l. This requires format specifications to be *complete* and *deterministic*. Completeness requires that for every possible bit string of length l, the format function returns exactly one of `true` or `false`. Determinism requires that any input to the format function will always be classified the same way. As a result, the format function defines a two-set partition of the space of bit strings of length l, with the two sets corresponding to `true` and `false` under the format, respectively. Taken together, the requirements of Delphinium ensure that at each iteration, the SAT formula will remain satisfiable as long as a query exists which can differentiate at least some `true` and `false` partition elements.

Challenge Intuition. Clearly, arbitrary functions may be more complex than predicates. To address this, a natural first idea is to implement a third possible response (in addition to `true` and `false`), e.g. `error`. By introducing an "error" class, outputs could be simplified and the completeness requirement removed. However, adding a third output class raises a unique challenge which the Delphinium authors leave to future work.

McFIL seeks to *simultaneously maximize* the number of secret inputs which correspond with *all* n output classes, such that each round, up to $\approx \frac{1}{n}$ of candidates may be eliminated in expectation. However, with the introduction of an error class and relaxed function definitions, the guarantee of simultaneous satisfiability is lost. For example, it may be that an input exists where some candidates would evaluate to a given output, and the rest to `error`, but none to another output. Delphinium requires simultaneous satisfiability of all classes, and so the resulting `UNSAT` result immediately halts progress.

This idea of three output classes is also only a specific instance of a much more general challenge. For arbitrary functions with n-bit outputs, predicates represent only $n = 1$. For functions with arbitrary output length $n > 1$, the number of simultaneously satisfiable classes is entirely dependent on function itself; as a result, naïve simultaneous maximization fails or reaches computational limits.

A First Step: `DeriveOutcomes`. Functions with n-bit outputs have a maximum of 2^n possible output values. However, for many functionalities the actual number of possible outputs is much lower. A natural example is a classification function: each class may be described by a long bit-string (a long output), but only a few classes may exist.

Detecting and exploiting these cases contributes to McFIL's practical applicability. At each iteration of an attack, the number of possible outputs defines the number of simultaneous maximization constraints which must be satisfied. Therefore, we provide the `DeriveOutcomes` subroutine, which uses the SAT solver to enumerate the possible outputs of the functionality. We perform this step one time at initialization (the *Configure Outcomes* step in Figure 2), dramatically reducing constraint system size from the 2^n maximum in many cases.

Analysis of `DeriveOutcomes`. `DeriveOutcomes` is executed once at the beginning of the McFIL workflow. It iterates over all 2^n possible outputs of the target n-bit output functionality in order to eliminate unreachable outputs. Although this can be concretely expensive, many functions with n-bit outputs do not entirely cover their range; eliminating large classes of unreachable outputs dramatically improves performance for the remainder of McFIL's steps. Additionally, this step can be skipped, and partial completion still benefits computation time significantly.

Optimizing Leakage: Straw-man Solution. With the set of possible outputs derived, it remains to eliminate large classes of candidates across these output classes at each iteration. Otherwise, the attack is approximately brute-force and therefore highly inefficient. If all $N \leq 2^n$ classes cannot be simultaneously differentiated from one another by a single query, the straightforward next step would be to differentiate *as many classes as possible*.

Here we reach the next challenge: choosing an optimal subset of output classes. The number of subsets of a set of N values is 2^N, i.e. 2^{2^n} for n-bit outputs. Even if `DeriveOutcomes` eliminates many outputs to reduce N, the exponential size remains problematic for performance. This combinatorial space is far too large to efficiently iterate through – worse, doing so would be required at *each iteration* due to evolving constraints. As a result, in order to support non-predicate functions which may contain mutually-exclusive outcome classes, optimality must be sacrificed in favor of an efficient heuristic which performs well in practice.

Solution Intuition. To avoid a combinatorial search, a heuristic must be employed. However, the accuracy of the heuristic directly impacts the profitability of the resulting query, and thus warrants specific analysis and consideration. The intuition for this heuristic (which we refer to as `SelectOutcomes`) is to remove outcomes from the simultaneous maximization constraint system while minimally affecting profitability.

To achieve this, we perform Model Counting (#SAT) on the solution space of each outcome individually. Each result corresponds to the maximum number of candidate models which *might be eliminated* by including that outcome in the simultaneous maximization. We sort all outcomes by their individual sizes, and remove them in ascending order until a simultaneously satisfiability subset is found.

Of course, this approach may miss an ideal configuration of outcomes. However, avoiding combinatorial search is necessary, and in our evaluation (§5), we demonstrate that profitability remains sufficient to discover efficient attacks.

Algorithm 2: `SelectOutcomes`

```
Input: Solver: SAT solver, Pool: multi-processing
       pool, $\mathcal{F}$: formula for functionality, $\mathbb{O}$: set of
       outcomes from DeriveOutcomes
Output: $\mathbb{O}^*$: set of mutually-compatible outcomes
        w/many candidate target, for use in
        maximization constraints
$\mathbb{O}^* \leftarrow \emptyset$;
for $i \leftarrow 1$ to $|\mathbb{O}|$ do
    $O_i \leftarrow i$th element of $\mathbb{O}$;
    Solver.constrain($O_i = \mathcal{F}$(chosen, target));
    // generate CNF for ApproxMC
    $\phi_i \leftarrow$ Solver.cnf();
    // remove constraint for next iteration
    Solver.pop();
    // call ApproxMC in parallel
    Pool.apply_async(ApproxMC, $\phi_i$);
// for each outcome and its ApproxMC count
for $(O_i, cnt)$ in Pool.results() do
    if $cnt > 0$ then
        // satisfiable single outcome
        $\mathbb{O}^*$.add(($O_i, cnt$));
$\mathbb{O}^* \leftarrow$ sort $\mathbb{O}^*$ by $cnt$ ascending;
while $|\mathbb{O}^*| > 1$ and not
 Solver.satisfiable($\mathbb{O}^*$) do
    $\mathbb{O}^*$.drop_first();
// largest simultaneously satisfiable set
return $\mathbb{O}^*$;
```

`SelectOutcomes`. To realize this heuristic, documented in simplified form in Algorithm 2, we employ a powerful tool from the recent SAT solving literature: ApproxMC [38]. ApproxMC ("Approximate Model Counting") is a tool which takes a CNF Boolean formula and rapidly provides an approximation of the formula's model count. ApproxMC can count complex formulae in a fraction of the time it takes to compute maximization constraints to count formulae. However, it cannot completely supplant maximization: ApproxMC is an external tool which uses sampling to provide approximate counts; there is no known way to efficiently encode iterative sampling *within the solver* as a constraint, and it is unlikely a method exists due to the data-dependent nature of the ApproxMC sample-and-iterate strategy.

By iterating through the remaining satisfiable outcomes at each iteration of the attack and invoking ApproxMC, we ensure that simultaneous maximization still occurs among as many large classes as can be efficiently identified. Further, we employ process-level parallelism to amortize this sequence of ApproxMC calls. ApproxMC configuration parameters can also trade off single-instance computation time for accuracy if needed. Once a set of mutually-compatible outcomes is found, the attack proceeds with simultaneous maximization using Max#SAT. The resulting query is extracted and executed in the to produce a result. By design, this result tends to eliminate a large class of remaining candidate solutions. In the event the result corresponds with a removed outcome class, relatively little knowledge is gained. However, as the removed outcome classes are the smallest, the likelihood that an arbitrary query induces a removed outcome is minimized over a distribution of possible `target` values.

Algorithm 2 documents the `SelectOutcomes` heuristic. In the first loop, each outcome is individually constrained to generate a set of formulae using our CNF manipulation interface and the underlying SAT solver, resetting the solver after each iteration to individually test each outcome. Parallel tasks are dispatched to perform `ApproxMC` counts of the bits corresponding to the `target` variable in each formula (bit correspondence requires formula manipulation, omitted for clarity). These results for each outcome are sorted, and until a simultaneously-satisfiable subset is found (more complex than a satisfiability check, but omitted for clarity), the outcomes with the smallest number of candidate `target` solutions are eliminated. Finally, the usable subset $\mathbb{O}^* \subseteq \mathbb{O}$ is returned.

Analysis of `SelectOutcomes`. Delphinium avoids the need for any such heuristics by strictly requiring well-defined and inflexible problem statements. As a result of these strict requirements, the authors of Delphinium are able to formally prove, probabilistically and approximately, a greedy-optimal algorithm. McFIL enables a far broader scope of functionalities to be analyzed and attacked. Further, reducing the restrictions on the formulae input to McFIL reduces the operator effort and expertise required. By allowing significantly more flexibility in terms of functionality choice and reducing the modeling work required, McFIL sacrifices formal optimality for generality, performance, and practicality.

4.2 Implementation

Algorithm 3: McFIL Algorithm Overview

```
Input: $\mathcal{F}$: Boolean circuit with target and chosen
       input(s), $\mathcal{O}$: Oracle access to functionality with
       hidden target input
Output: $\mathcal{I}^*$: set of Boolean circuit inputs (vectors of
        Booleans) for chosen which maximize
        leakage of target in $\mathcal{F}$
$\mathcal{I}^* \leftarrow \emptyset$;
$out \leftarrow$ DeriveOutcomes($\mathcal{F}$);
while # of solutions for target > 1 do
    $\overrightarrow{sel} \leftarrow$ SelectOutcomes($\mathcal{F}, out$);
    $query \leftarrow$ Maximize(chosen, $\mathcal{F}, \overrightarrow{sel}$);
    $result \leftarrow \mathcal{O}(query)$;
    $\mathcal{F}$.AddConstraint($\mathcal{F}(query$, target$) = result$);
    $\mathcal{I}^*.add(query)$;
return $\mathcal{I}^*$;
```

McFIL consists of under 2 KLoC of new Python3 which leverages process-level parallelism at every opportunity. We introduce a CNF translation layer to readily convert SAT instances and even individual CNF clauses and literals between the two SAT solvers we employ, CryptoMiniSat and Z3, to reap the relative benefits of each (performance and flexibility, respectively). Our implementation includes a test shim for simulated execution of protocols, and a convenient command-line interface encapsulating numerous configuration options. It is available as open source software on GitHub [49]. Algorithm 3 provides a high-level pseudo-code overview of McFIL using the subroutines described in Section 4.1.

CryptoMiniSat. CryptoMiniSat [39] (CMS) by Soos *et al.* is a SAT solver designed for cryptographic use-cases. Specifically, CMS includes optimizations for rewriting and processing *Xor* operations which otherwise incur exponential overhead in the number of operands of a CNF representation. Specifically, an n-term *Xor* expands to 2^{n-1} CNF clauses.

We evaluate McFIL in §5 and provide wall-clock computation time to illustrate the practicality of attack generation when using CMS. We confirm the observation of Beck *et al.* that SAT instances containing *Xor*-dense maximization constraints execute up to an order of magnitude faster in CMS than Z3, and for larger (longer-running) problem instances our parallelized query search offers additional multiplicative factors of time savings.

Z3. Z3 [16] is an SMT solver which provides extensive and robust software support for Boolean formula manipulation

Table 1: Evaluated Functionalities

Functionality	`target`	**Leakage**
Yao's Millionaires	2^{64}	$2^{30\pm1}$
Dual Execution	2^{12}	$2^{10} - 2^{11}$
Danish Sugar Beets Auction	$2^{14} - 2^{56}$	up to 2^{54}
Bucketed Mean	2^{32}	$2^{23\pm1}$
Wage – Circuit Division	2^{36}	$2^{34\pm1}$
Wage – Standard Division	2^{36}	$2^{34\pm1}$
Mean Average	2^{8}	$2^{0} - 2^{1}$

Refer to §5.1. Smaller domain sizes (`target`) were chosen to allow many randomized trials. Leakage conservatively estimates eliminated candidates per query, calculated in a single iteration of McFIL.

Table 2: CNF Size in Clauses over Target Bits

Functionality	**8-bit**	**16-bit**	**32-bit**	**64-bit**
Yao's Mill.	47	95	191	383
Dual Exec.	875	3539	14243	57155
*Danish.**	2746	6500	8701	14020
BM	457	561	769	1185
WCD	**	**	2178	5351
WSD	60	749	7705	42417
MA	191	399	815	1647

Refer to overview and descriptions in §5.1.

**Due to encoding, the Danish Sugar Beets functionality was measured at 12, 28, 42, and 60 bits.*

***The BWWC WCD function reorganizes division into a multiplication circuit which overflows for small bit-widths.*

and solving. It is a general-purpose solver which supports arithmetic, bit-vector, and other common theories for ease of use in translating general problems to SAT. We use Z3 for its bit-vector theory support and other tooling. However, Z3 lacks key optimizations which accelerate solving the the particular Xor-dense maximization formulae we require. As a result, after formula preparation in Z3, we generate and export a CNF and use CMS for solving, and then recover results back into Z3 for the next round of constraint manipulation.

5 Evaluation

To evaluate McFIL, we assess our implementation against a range of functionalities, from simple motivating examples such as the classic Yao's Millionaires problem [47] to the recent practical instantiation of MPC developed by researchers at Boston University with the Boston Women's Workforce Council (BWWC) to measure wage equity in a manner which preserved the privacy of participants' salaries [26]. The success of McFIL in partially or completely deriving confidential inputs across these functionalities demonstrates its usefulness as a tool for practitioners and researchers alike in performing privacy analysis of secure protocols.

5.1 Selected Functionalities

The following selected target functionalities are used to evaluate McFIL. Implementations of these functionalities are provided in the open source release using a simple Python3 domain-specific language to describe Boolean formulae. Samples of this DSL can be found in Appendix A. We provide additional illustrative samples used in our evaluation in our open source release [49]. These implementations may be useful for further analysis, or to serve as templates for Boolean formula representations of new functionalities.

Evaluated Functionalities. Table 1 lists the evaluated functionalities and the search space size of each corresponding hidden `target` value. Relatively smaller search spaces were chosen compared with what is practically achievable in reasonable wall-clock time. This allowed evaluation of computation bottlenecks in McFIL while keeping overall evaluation runtime feasible for many randomized repetitions. McFIL outputs estimated leakage per query, useful for evaluating functionalities after only a single iteration of the tool. CNF size growth (in # *clauses*) across input sizes is demonstrated in Table 2 to inform extrapolations.

5.1.1 Millionaires Problem

The Millionaires problem introduced by Yao [47] describes two millionaires, Alice (A) and Bob (B), who wish to compare their wealth without revealing it. Each has a corresponding input, their net worth represented as 64-bit integers a and b, respectively. The functionality to be computed in this simple example is the comparison operation less-than. Thus, at the end of the two-party computation, A and B learn the result of the predicate $a < b$, but learn nothing of each other's input.

This functionality clearly enables a binary search, and the resulting exponential decreases in the search space per query confirm the attack algorithm's approximate optimality (§5.2). We implement this functionality within `millionaires.py`. The attack is not particularly subtle, and could certainly be developed and executed manually. However, the example is illustrative, demonstrating the SAT solver rediscovering the known optimal attack without interaction or guidance.

5.1.2 Dual Execution

Dual Execution [30] is an MPC technique which enables conversion of semi-honest secure protocols into malicious secure-with-abort protocols incurring only a single bit of additional leakage. The technique involves both parties in a 2PC garbling a Boolean circuit and sending each other the circuit

Table 3: Evaluation Results

Functionality	`target`	**Size**	**Queries**	**Mean**	**S.D.**	**Avg. Time**	**Parallel Speedup**
Yao's Millionaires	●	2^{64}	69 – 97	84.7	8.5	≈ 3 mins	≈ 2×
Dual Execution (affine)	●	2^{12}	17 – 80	34.9	12.9	≈ 0.5 mins	≈ 1.5×
Danish Sugar Beets Auction	◐	2^{28}	13 – 13	13.0	0.0	≈ 5 mins	≈ 4 – 5×
BWWC Bucketed Mean	●	2^{32}	35 – 58	36.7	6.4	≈ 6 mins	≈ 10×
BWWC Wage (circuit div.)	◐	2^{36}	19 – 26	22.3	5.0	≈ 0.5 mins	≈ 1.5 – 2×
BWWC Wage (standard div.)	◐	2^{36}	18 - 29	22.4	5.8	≈ 5 mins	≈ 1.5 – 2×
BWWC Mean Average	●	2^{8}	2 – 3	2.5	0.5	≈ 2.5 mins	≈ 0.8 – 1.2×

`target` uncovered ...partially ◐ ...completely ●

Size denotes the size of the `target` domain. Average time given for full attack, not per-query. Minimum and maximum queries listed with Mean and Standard Deviation. Average queries reported over ≈ 100 randomized trials. 'Speedup' denotes wall clock time savings through parallelization.

and necessary data for its evaluation. Both circuits are evaluated, and then a secure comparison protocol informs both parties if the outputs matched. If they do not, the protocol is aborted. The additional bit of leakage comes from the abort, which informs an adversary that the garbled circuit it sent to the honest party did not match the output of the one generated by the honest party – this knowledge can be leveraged to leak a bit of the honest party's private input upon each execution.

To implement this part of the adversary's input string is considered to be an encoded program. This program represents the divergent functionality the adversary may choose. For simplicity, we limit the adversary to affine transformations, which can be realized through a matrix multiplication. The complexity of function is entirely up to the adversary, however, allowing configurability is preferable in the dual execution setting where the honest party must be expected to believe they are executing an honest circuit rather than expecting an abort due to a mismatched equality check. The adversary then provides both its own input and the "code" the honest party will execute. McFIL is able to uncover all bits of the private input in very few queries (§5.2), even with the limitation to affine functions.

5.1.3 Danish Sugar Beets Auction

The first widely-known practical use of MPC was reported in 2009, when the Danish sugar beets auction was deployed as an MPC. The purpose was to replace the work of a trusted-by-necessity auctioneer with secure computation. The functionality takes in a set of buyer and seller orders to determine the *Market Clearing Price*. An order consists of a set of prices and the number of units (e.g. tons of sugar beets) a buyer/seller is willing to buy/sell at each price. The Market Clearing Price is the equilibrium price at which optimal volumes of sugar beets are exchanged. The functionality is characterized by a matching of buyer and seller prices weighted by the units at each price.

In this protocol, there are a configurable number of buyers and sellers, and we leverage this to test McFIL in the multi-party MPC setting (rather than two-party computation). When modeling adversarial participants, McFIL expresses all adversary inputs as a single vector of Boolean variables over which it determines structure and exploits leakage based on the functionality (Market Clearing Price calculation). This encodes the well-understood behavior of colluding malicious participants in an MPC. We implement this within `sugarbeets.py`.

5.1.4 BWWC

Collaborating with the Boston Women's Workforce Council (BWWC), Lapets *et al.* at Boston University recently developed and deployed an MPC system to detect gender-based wage inequity [26]. This system allowed companies to privately share aggregate statistics about employee roles, salaries, and gender. The goal was to computationally identify and measure gender-based wage discrimination without requiring any participating companies to directly demonstrate fault. Their design, robustly proven secure under standard assumptions, is an MPC which computes a simple statistical test among each comparable role across companies. By design, their system protects the privacy of each salary data point.

For this target, we developed two aggregation functions inspired by the functionality described. The first was a bucketed mean computation (`mean_buckets.py`) which averaged the two inputs and then returned a result representing the number of fixed-size buckets apart the query is from the updated mean. This bucketing procedure implements a non-predicate functionality in which many outcomes are simultaneously satisfiable, exercising the attack algorithm in this regard. The second averages two contributed salary data points (one honest input, one adversary) into a large aggregated salary, and then informs the querying adversary if their contributed salary

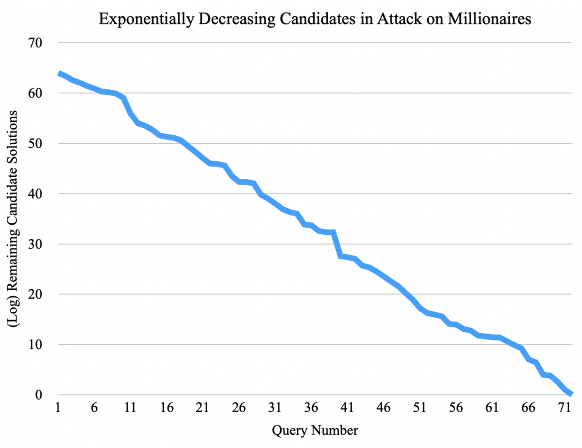


Figure 4: Remaining candidate solutions per query in a single trial of 64-bit Yao's Millionaires

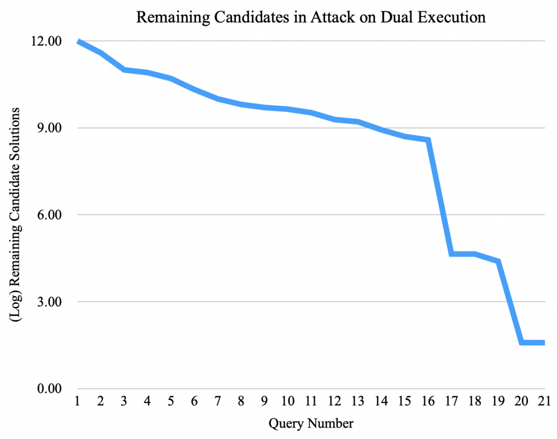


Figure 5: Remaining candidate solutions per query in a single trial of Dual Execution (Affine Predicates)

is below the updated mean. This functionality enables the attack algorithm to derive a binary search for the honest input as demonstrated in the results section (§5.2).

We developed two forms of this second function, one (`wage_circuit_div.py`) with the division reorganized into a multiplication in the mean computation, and the other (`wage.py`) with regular integer division inside the solver. Division steps dominate the complexity of this functionality, as demonstrated by the difference between BWWC functions 2 and 3 in Table 1. Finally, we implemented a plain mean average (`mean.py`) function (BWWC function 4 in the Table) to evaluate a function with maximal (2^n) possible outputs for an n-bit `target`. Despite the small size of the secret in this case, computation time increased due to the large output domain as demonstrated later in this section.

5.2 Evaluation Results

In this section, we evaluate McFIL by measuring its effectiveness in uncovering private `target` input(s) in the selected functionalities. Table 3 provides a summary of evaluation from repeated trials with random `target` values.

Process-Level Parallelism. McFIL uses parallelism in two key subroutines, `SelectOutcomes` and in computing the `chosen` query at each iteration. For an n-bit `target`, `SelectOutcomes` spawns one process per possible outcome (up to 2^n), and computing the query replaces an $O(n)$ linear parameter sweep from Delphinium with n parallel solving instances. For details on hardware, refer to Appendix B.

5.2.1 Millionaires Problem

As previously discussed, the Yao's Millionaires functionality consists of a comparison between the adversary and hidden inputs. This enables a binary search which, as seen in Figure 4, eliminates approximately half of the remaining candidates at each iteration. The linear descent of the graph on a logarithmic scale captures the efficiency of the attack, resulting in approximately $log(n)$ queries for an n-bit `target`.

Parallelism in the attack generation pipeline achieves an approximately $2\times$ speedup in the Millionaires functionality. Savings occur in the parallel ApproxMC of the two outcomes in the `SelectOutcomes` heuristic. In this case, however, the heuristic is not required as the comparison predicate is complete and the two outcomes are not mutually exclusive. As a result, McFIL incurs additional overhead in exchange for its ability to handle mutually-exclusive outcome classes. We have added software arguments to configure (optimize) McFIL when the functionality is known to be complete.

5.2.2 Dual Execution

In this attack, McFIL generates a sequence of matrices and inputs in order to rule out candidate `target` inputs. Effectively, this attack creates an increasingly large system of linear equations choosing the coefficients and half the unknowns (`chosen`) at each step.

Figure 5 demonstrates the approximately logarithmic decreases of the remaining `target` search space over the course of an iterative attack. The plateaus visible in the graph further support the idea that there may be points at which McFIL is effectively "guessing and checking" within the available information, and once these guesses succeed (after 2-3 attempts, in the visualized attack) the following queries are able to eliminate a large quantity of candidate solutions.

5.2.3 Danish Sugar Beets Auction

In the Danish Sugar Beets Auction functionality, not all `target` bits are uncovered. Figure 6a in Appendix B denotes the rapid descent of remaining candidate solutions on a log scale, however, the attack does not reach $2^0 = 1$ (a unique solution). This seems to result purely from insufficient leakage (or, sufficient privacy) of the functionality. At the extremes of low and high prices, McFIL is able to determine the honest buyers/sellers bids for numbers of units. However, at middle prices closer to the likely equilibrium Market Clearing Price, McFIL fails to distinguish and some bits remain unknown.

In this example, we configured the adversary to control 3 buyers and 3 sellers, and the honest party to control 1 buyer and 1 seller. The adversary's goal (and therefore McFIL's `target`) was to uncover the prices and numbers of units of the honest parties' orders. Although only partial knowledge is attained, the security model of the MPC [5] treats the entire prices/amounts list as confidential, and so this attack still represents a meaningful compromise of the intended privacy. Additional parameters and configurations of malicious and honest parties are explored in Appendix B.

5.2.4 BWWC

We evaluate three functionalities derived from the Boston Women's Workforce Council MPC [26]: Bucketed Average, Wage Equity (with two variants), and Mean Average.

Bucketed Average. In the Bucketed Average functionality, McFIL is only able to uncover the `target` to the granularity of the buckets (ranges). By definition of the functionality, averages which land in the same buckets are indistinguishable, and so this result is expected. Figure 6b in Appendix B demonstrates the progress of the iterative attack, noting that it completes before finding a unique solution for `target`.

Due to the significant number of outcome classes, the `SelectOutcomes` subroutine dominates execution time. Specifically, the CNF generation step prior to parallelization incurs the most overhead. We discuss the overhead of CNF generation in §6. Once CNFs have been exported, parallelized ApproxMC may be employed, and due to the significant number of independent ApproxMC tasks the speedup in this case achieves up to $10\times$ at some iterations.

Wage Equity. In both the *circuit division* and *standard division* variants of the functionality, the wage-average computation leaks significant information of the high-order bits of the honest party's input salary. Although the salary is not uncovered in full, learning the most significant upper half of this value effectively defines the salary range – disclosing information intended to be private.

In the *circuit division* instance, the mean calculation is reorganized to use a multiplication inside the solver rather than an unsigned integer division. Integer division corresponds with a complex Boolean formula after translation through Z3's SMT interface into SAT due to the edge-case handling of division. By applying some basic mathematical insight to adapt the target functionality to be more amenable to SAT, we achieve a notable increase in performance.

The performance gain is evident compared to *standard division*. As noted in Table 3, the overall computation time at a given bit width differs by an order of magnitude. However, as indicated by the similarity of Figure 6c and Figure 6d in Appendix B, the resulting attacks achieve similar *per-query* efficiency in eliminating candidate solutions.

Both functionalities encode predicates with two simultaneously satisfiable outcomes. As a result, the `SelectOutcomes` heuristic is unneeded, and as a result is pure overhead lost in exchange for the assurance that the attack will proceed even if the outcomes are or become mutually incompatible. However, parallelism minimizes the impact of this overhead attaining a $1.5-2\times$ speedup across the two outcomes.

Mean Average. In the Mean Average functionality, the output range of the functionality is the full domain of its inputs. This functionality takes two inputs and computes their mean. As the inputs are bitvectors rather than purely mathematical integers or reals, some complexity is introduced to this averaging through floored division and the 2^n-1 bound of an n-bit value. As a result, McFIL finds attacks which generally require two queries, occasionally three.

Although this relatively simple functionality admits a query-efficient attack, it is a useful benchmark due to the maximal number of possible outputs. The full-domain output (of exponential size in n) taxes `SelectOutcomes` to the maximum degree per `target` bit. As a result, the bottleneck of McFIL is the `SelectOutcomes` heuristic, specifically in CNF generation for each SAT instance. CNF generation via the Tseitin transformation [42] occurs within the Z3 SMT solver and, while asymptotically efficient, can require significant computation time. Unfortunately, as CNF generation vastly exceeds ApproxMC solving time within `SelectOutcomes`, process-parallelism offers little benefit and even occurs slight overhead in some tests.

Mean Average-like functionalities are the key motivator for the `SelectOutcomes` heuristic. With a Delphinium-like approach, the attack immediately fails with an UNSAT result. The reason for this failure becomes clear by example.

Consider a 2-bit adversary input a, and 2-bit honest input b. $\texttt{mean}(a,b) \in \{0,1,2,3\}$. Configuring simultaneous maximization to find an optimal a, the following constraint (among others) is added to the solver. The symbolic representations of the four partitions of b denoted $\{b_0,b_1,b_2,b_3\}$ correspond to b when the mean with a is 0, 1, 2, and 3, respectively.

$$mean(a,b_0)=0 \,\wedge\, mean(a,b_1)=1 \,\wedge\\ mean(a,b_2)=2 \,\wedge\, mean(a,b_3)=3$$

Notice that the first clause of the conjunction requires (a,b_0) to be $(0,0)$, $(0,1)$, or $(1,0)$ (with floored division). Therefore $a \in \{0,1\}$. However, the last clause requires $(a,b_3)=(3,3)$ exclusively. As $\{0,1\} \cap \{3\} = \emptyset$, the conjunction is unsatisfiable. `SelectOutcomes` correctly identifies that b_1 and b_2 are not similarly mutually exclusive, and that they contain the largest number of candidate solutions for b. As a result, b_0 and b_3 are selected out, and the attack proceeds without interruption. McFIL is then able to derive an optimal or near-optimal attack in 2 – 3 queries.

5.3 Contributed Benchmarks

As an additional artifact, we have submitted a wide assortment of benchmarks to the SMT-LIB compendium of SAT benchmarks [3]. These have been accepted to help aid solver development and increase performance on the particular instance types McFIL encounters. Advancements in SAT research can offer a drop-in improvement to this work by expanding the horizon of computational feasibility. A summary of the submitted files is presented in Appendix C. A subset of these benchmarks have already begun to see use within SAT research as a basis to test a new model counting approach [35].

6 Discussion

Our results demonstrate that McFIL is capable of evaluating a diverse array of simple functionalities in a matter of minutes of computation time. The generated attacks, which rely on the aforementioned multi-run adaptive assumption, are able to uncover all bits of the private `target` value in many cases. When our approach cannot continue to differentiate classes of candidate solutions to eliminate, it notifies the user that the attack may devolve to "brute-force" and asks if they'd like to continue. This occurs when `SelectOutcomes` fails to find a simultaneously satisfiable set of outcomes.

We observe that predicate functionalities execute relatively quickly, and functionalities with larger outputs (e.g. the Danish sugar beets auction and bucketed mean functionalities) spend significant time in `SelectOutcomes`. On the extreme end, the mean average functionality has an output domain as large as the output bitvector allows – and we correspondingly observe the most time spent in navigating these highly mutually-exclusive outcome classes.

In multiple cases, the `SelectOutcomes` heuristic is the computational bottleneck for attack progression. Analyzing these cases more closely reveals that the CNF generation step, using Tseitin's method [42] to generate a CNF from an arbitrary Boolean formula with only linear expansion, is the crux of the subroutine. Despite its asymptotic efficiency, this step (as executed within Z3) takes significant time. We employ a number of CNF manipulation and caching techniques to avoid unnecessary regeneration of CNFs where possible, however, the relative speed of parallelized ApproxMC leaves this sequentially-executed step as the longest-running component.

Even without the multi-run adaptive assumption to allow an iterative attack, the initial analysis step quantifies approximately how much leakage can be exploited with a given optimized query. On its own, this enables privacy/confidentiality analysis of any functionality planned for inclusion into an MPC, FHE, or ZK scheme. Further, this analysis needs to be run only once for a given functionality, and so even if hours or days of computation are required for some complex functionality, the resulting quantified leakage or iterative attack can provide pivotal insights to researchers and developers.

Limitations. The remaining practical limitations of McFIL are largely entangled with the inherent complexity of the computational problems it employs. For sufficiently complex functionalities or large output domains, running time increases significantly. Depending on the use-case, this may not rule out the use of McFIL for important/sensitive functionalities or contexts. Notably, for functionalities with very large output domains, or which encode cryptographic primitives (e.g. the AES round function [24]) directly within their Boolean circuitry, McFIL reaches wall-clock performance bottlenecks which may impede its applicability.

McFIL seeks to reduce operator burden and required expertise to develop confidentiality attacks or measure leakage. However, a certain degree of operator involvement is still required: McFIL can only be as accurate as the Boolean formula representation of a target functionality. The Python3 DSL we provide is an initial step in this direction, however it still requires understanding and manual effort.

7 Conclusion

SAT and cryptography have intersected numerous times in the research literature, often to their mutual benefit. McFIL pursues this interdisciplinary exchange by applying emerging SAT techniques to a new domain, bridging theory and practice, and contributing back to the SAT community. With McFIL, developers of secure protocols can automatically determine privacy thresholds or generate attacks against candidate systems. Potential users of these tools can evaluate them before choosing to use them. For sufficiently sensitive use cases, extensive computation times for complex or large functionalities may be worthwhile; McFIL empowers practitioners to make that decision without requiring expensive expert analysis.

Acknowledgments

We would like to thank the anonymous reviewers and Shepherd for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361, a DARPA Young Faculty Award (YFA) under Grant Agreement D22AP00137-00, and by DARPA under Contract No. HR001120C0084. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government, the NSF, or DARPA. The authors would also like to thank Katarina Mayer for her contributions to this work.

References

[1] Ghada Almashaqbeh, Fabrice Benhamouda, Seungwook Han, Daniel Jaroslawicz, Tal Malkin, Alex Nicita, Tal Rabin, Abhishek Shah, and Eran Tromer. Gage mpc: Bypassing residual function leakage for non-interactive mpc. *Proceedings on Privacy Enhancing Technologies*, 4:528–548, 2021.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. Enforcing ideal-world leakage bounds in real-world secret sharing mpc frameworks. In *IEEE CSF '18*, pages 132–146, 2018.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Automating the development of chosen ciphertext attacks. In *USENIX Security '20*, pages 1821–1837, 2020.

[5] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343, 2009.

[6] Elette Boyle, Shafi Goldwasser, Abhishek Jain, and Yael Tauman Kalai. Multiparty computation secure against continual memory leakage. In *ACM STOC '12*, pages 1235–1254, 2012.

[7] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. Sok: Design tools for side-channel-aware implementations. In *ACM AsiaCCS '22*, pages 756–770, 2022.

[8] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P '18*, pages 315–334. IEEE, 2018.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Supratik Chakraborty, Daniel Fremont, Kuldeep Meel, Sanjit Seshia, and Moshe Vardi. Distribution-aware sampling and weighted model counting for sat. In *AAAI '14*, volume 28, 2014.

[11] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. Technical report, National University of Singapore, 2016.

[12] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference: Information theory and information flow. In *Workshop on Issues in the Theory of Security (WITS'04)*, 2004.

[13] Michael R Clarkson, Andrew C Myers, and Fred B Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.

[14] Stephen A. Cook. The complexity of theorem proving procedures. In *ACM STOC '71*, pages 151–158, 1971.

[15] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS '08*, volume 4963 of *LNCS*, pages 337–340, 2008.

[17] Daniel Fremont, Markus N. Rabe, and Sanjit A. Seshia. Maximum Model Counting. In *AAAI '17*, February 2017.

[18] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC '09*, pages 169–178, 2009.

[19] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[20] Carmit Hazay, Abhi Shelat, and Muthuramakrishnan Venkitasubramaniam. Going beyond dual execution: Mpc for functions with efficient verification. In *IACR PKC '20*, pages 328–356, 2020.

[21] Holger H Hoos and Thomas Stützle. Satlib: An online resource for research on sat. *Sat*, 2000:283–292, 2000.

[22] Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE S&P '12*, pages 272–284, 2012.

[23] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *ACM STOC '07*, pages 21–30, 2007.

[24] Abdel Alim Kamal and Amr M Youssef. Applications of sat solvers to aes key recovery from decayed key schedule images. In *2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, pages 216–220. IEEE, 2010.

[25] Vladimir Klebanov, Norbert Manthey, and Christian Muise. Sat-based analysis and quantification of information flow in programs. In *International Conference on Quantitative Evaluation of Systems*, pages 177–192, 2013.

[26] Andrei Lapets, Nikolaj Volgushev, Azer Bestavros, Frederick Jansen, and Mayank Varia. Secure mpc for analytics as a web application. In *IEEE SecDev '16*, pages 73–74, 2016.

[27] Piotr Mardziel, Michael Hicks, Jonathan Katz, Matthew Hammer, Aseem Rastogi, and Mudhakar Srivatsa. Knowledge inference for optimizing and enforcing secure computations. In *Proceedings of the Annual Meeting of the US/UK International Technology Alliance*, 2013.

[28] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies. In *IEEE CSF '11*, pages 114–128, 2011.

[29] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, 21(4):463–532, 2013.

[30] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In *International Workshop on Public Key Cryptography*, pages 458–473, 2006.

[31] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of Adaptive Side-Channel Attacks. In *IEEE CSF '17*, 2017.

[32] Python Software Foundation. Process pools. https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.pool, 2022.

[33] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A Hammer. Knowledge inference for optimizing secure multi-party computation. In *ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 3–14, 2013.

[34] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE S&P '14*, pages 459–474, 2014.

[35] Arijit Shaw. Research statement. https://www.tcgcrest.org/wp-content/uploads/2022/02/Arijit.pdf. Private communications with author.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P '16*, 2016.

[37] Joao P Marques Silva and Karem A Sakallah. Grasp-a new search algorithm for satisfiability. In *ICCAD*, volume 96, pages 220–227, 1996.

[38] Mate Soos and Kuldeep S. Meel. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *AAAI '19*, 2019.

[39] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 244–257, 06 2009.

[40] Larry J. Stockmeyer. The polynomial-time hierarchy. *TCS '76*, 3(1):1–22, 1976.

[41] Seinosuke Toda. PP is As Hard As the Polynomial-time Hierarchy. *SIAM J. Comput.*, 20(5):865–877, October 1991.

[42] Grigorii Samuilovich Tseitin. On the complexity of proof in prepositional calculus. *Zapiski Nauchnykh Seminarov POMI*, 8:234–259, 1968.

[43] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[44] Leslie G Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[45] Dan Walters, Andrew Hagen, and Eric Kedaigle. Sleak: A side-channel leakage evaluator and analysis kit. Technical report, MITRE CORP BEDFORD MA, 2014.

[46] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[47] Andrew C Yao. Protocols for secure computations. In *SFCS '82*, pages 160–164, 1982.

[48] Andrew C Yao. How to Generate and Exchange Secrets. In *SFCS '86*. IEEE Computer Society, 1986.

[49] Maximilian A. Zinkus. McFIL-Release on GitHub. https://github.com/maxzinkus/McFIL-Release, 2023.

A Solver DSL Samples

Python3 DSL example for Yao's Millionaires:

```
def func_smt(solver, chosen_input, target_input):
    """ Millionaire's functionality inside the solver """
    return solver._if(solver._ugt(chosen_input, target_input), # Millionaire's
                      solver.bvconst(1,OUTCOME_LEN),
                      solver.bvconst(0,OUTCOME_LEN))

def func(chosen_input, target_input):
    """ Millionaire's functionality outside the solver """
    return chosen_input > target_input
```

Python3 DSL example for Dual Execution (affine predicates):

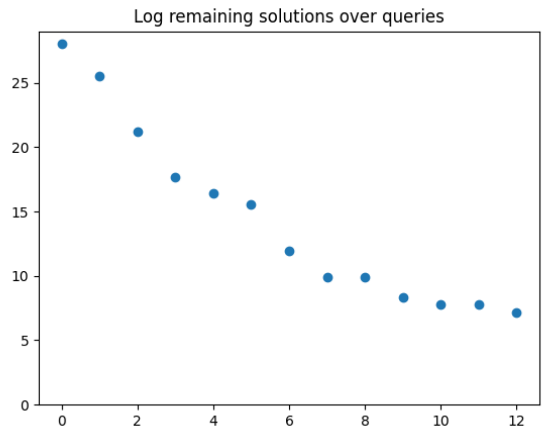
```
def func_smt(solver, chosen_input, target_input):
    # isolate adversary-chosen function (matrix) from adversary-chosen input
    matrix_bits = solver.extract(chosen_input, CHOSEN_LEN-1, TARGET_LEN)
    chosen_input = solver.extract(chosen_input, TARGET_LEN-1, 0)
    # unpack matrix values
    matrix = [[solver.extract(matrix_bits, j*TARGET_LEN+i, j*TARGET_LEN+i)
              for i in range(TARGET_LEN)] for j in range(TARGET_LEN)]
    chosen_bits = [solver.extract(chosen_input, i, i) for i in range(TARGET_LEN)]
    target_bits = [solver.extract(target_input, i, i) for i in range(TARGET_LEN)]
    # perform mults
    chosen_matrix = [reduce(lambda x, y: solver._add(x, y),
                            [solver._mult(matrix[j][i], chosen_bits[i])
                             for i in range(TARGET_LEN)])
                     for j in range(TARGET_LEN)]
    target_matrix = [reduce(lambda x, y: solver._add(x, y),
                            [solver._mult(matrix[j][i], target_bits[i])
                             for i in range(TARGET_LEN)])
                     for j in range(TARGET_LEN)]
    # re-pack matrices
    chosen_out = solver.concat(*reversed(chosen_matrix))
    target_out = solver.concat(*reversed(target_matrix))
    # evaluate equality check
    return solver._if(solver._eq(chosen_out, target_out),
                      solver.bvconst(1,1),
                      solver.bvconst(0,1))

def func(chosen_input, target_input):
    matrix_bits = chosen_input >> TARGET_LEN
    chosen_input = chosen_input & ((1 << TARGET_LEN)-1)
    matrix = [[(matrix_bits >> (i+TARGET_LEN*j)) & 1 for i in range(TARGET_LEN)]
             for j in range(TARGET_LEN)]
    chosen_bits = [(chosen_input >> i) & 1 for i in range(TARGET_LEN)]
    target_bits = [(target_input >> i) & 1 for i in range(TARGET_LEN)]
    chosen_matrix = [sum([(matrix[j][i]*chosen_bits[i])%2 for i in range(TARGET_LEN)])
                        % 2
                     for j in range(TARGET_LEN)]
    target_matrix = [sum([(matrix[j][i]*target_bits[i])%2 for i in range(TARGET_LEN)])
                        % 2
                     for j in range(TARGET_LEN)]
    chosen_out = 0
    for i, bit in enumerate(chosen_matrix):
        chosen_out |= bit << i
    target_out = 0
    for i, bit in enumerate(target_matrix):
        target_out |= bit << i
    return 1 if chosen_out == target_out else 0
```

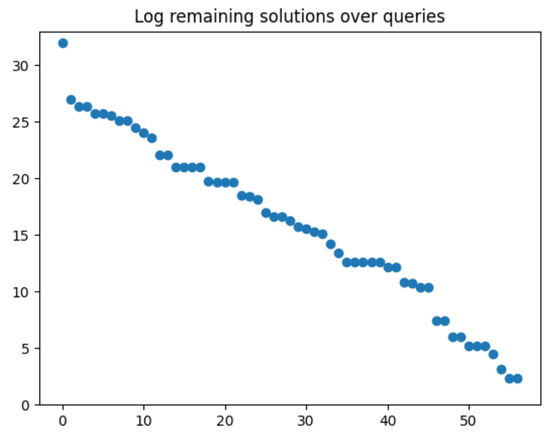
B Additional Evaluation

Each graph within Figure 6 below denotes a representative example attack generated by McFIL. Each test operates over a domain of 28- to 36-bit values in order to keep overall benchmarking time reasonable with many repetitions of each attack. Some attacks do not trend to 2^0 as the functionality does not admit complete leakage of the underlying secret (honest) input(s).

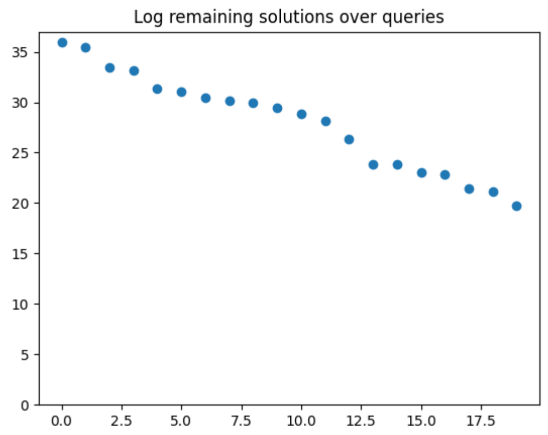
Testing Environment. All evaluations were performed on an Intel Xeon E5 CPU at 2.10GHz with 500GB RAM. Process-level parallelism was configured to employ 64 of the available virtual threads of execution using a Python3 standard library process



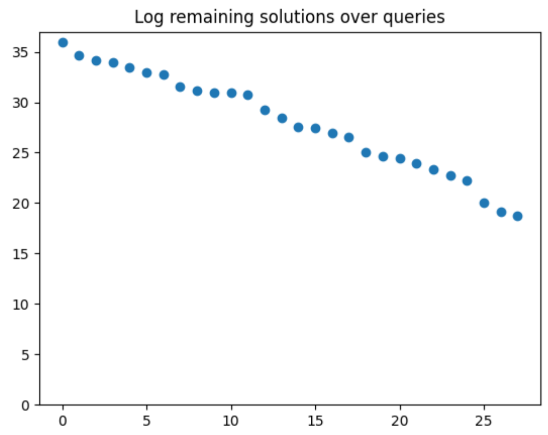
(a) Danish Sugar Beets Auction (2^{28})



(b) BWWC Bucketed Mean (2^{32})



(c) BWWC Wage (circuit division) (2^{36})



(d) BWWC Wage (standard division) (2^{36})

Figure 6: Remaining candidate solutions (log scale) per query

pool [32]. We used CryptoMiniSat 5.8.0, ApproxMC 4.0.1, Z3 4.8.15, and Python3 3.8.10.

Many-Party MPC in the Sugar Beets Auction. The Sugar Beets Auction functionality demonstrates McFIL in the setting where more than two MPC participants are involved. In effect, McFIL treats multi-party protocols as 2PC: the solver is simply aware of bits that are symbolic and out of its control, and bits that it is able to vary to induce satisfiability. However, as it is an illustrative example, we demonstrate the varying results of McFIL as the number of honest and colluding malicious parties varies. Often, the uncovered `target` bits does not completely cover the domain, but in these cases McFIL generally uncovers one or more high bits of each Honest party's input, restricting their possible values to smaller ranges within the domain.

C Contributed Benchmarks

Throughout the evaluation of McFIL, numerous complex SAT formulae were generated and stored as text files. The SAT community aggregates such files to serve as benchmarks and testing tools for international competitions of SAT solver speed and capability. Table 5 summarizes the benchmark CNF files derived from each functionality and characterizes them in terms of clause and variable counts given as ranges and an average across all files. These CNFs have been accepted to the SMT-LIB benchmark collection [3] used in SAT solver competitions which motivate research and improvement. Each CNF instance is paired with an equivalent SMT2 file defined in the quantifier-free bitvector (`QF_BV`) domain; these equivalent but differently

Table 4: Additional Sugar Beets Auction Evaluation

Functionality	`target`	Domain Size	Queries	Avg. Time	Parallel Speedup
4 Players – All Sellers Malicious	14 – 18	2^{28}	11 – 15	$\approx$ 5 mins	$\approx 4-5\times$
4 Players – All Buyers Malicious	8 – 18	2^{28}	8 – 15	$\approx$ 5 mins	$\approx 4-5\times$
4 Players – Half Each Malicious	12 – 12	2^{28}	11 – 18	$\approx$ 5 mins	$\approx 4-5\times$
4 Players – One Buyer Malicious	0	2^{42}	6 – 10	$\approx$ 3 mins	$\approx 4-5\times$
4 Players – Three Buyers Malicious	14	2^{14}	8 – 9	$\approx$ 1 mins	$\approx 2\times$
6 Players – 2 Malicious 1 Honest Each	10 – 14	2^{28}	11 – 12	$\approx$ 5 mins	$\approx 2-4\times$
6 Players – 1 Malicious 2 Honest Each	8 – 20	2^{56}	12 – 20	$\approx$ 20 – 30 mins	$\approx 1-7\times$

`target` bits discovered (maximum $log_2(domain)$)

Average time given for full attack, not per-query. Minimum and maximum queries listed.
'Speedup' denotes wall clock time savings through parallelization.

encoded files have proven useful in SAT research alone and in contrast to their CNF pairs [35]. Finally, in the course of generating, gathering, and testing numerous randomized SAT instances, a variety of underlying software bugs in the CryptoMinisat [39] and Z3 [16] solvers were uncovered, reported to open-source software maintainers, and resolved collaboratively.

Table 5: Benchmarks Overview

Functionality	Instances	CNF Clauses	Avg. Clauses	CNF Variables	Avg. Variables
Yao's Millionaires	1,504	1,158 – 62,676	17,116	506 – 17,039	4,524
Dual Execution (affine)	729	4,898 – 49,160	8,562	1,549 – 12,375	2,290
Danish Sugar Beets Auction	88	49,020 – 159,349	112,178	10,451 – 27,640	19,834
BWWC Bucketed Mean	345	29,794 – 66,138	46,238	8,824 – 17,036	12,593
BWWC Wage (circuit div.)	194	6,058 – 11,053	7,655	1,283 – 2,465	1,571
BWWC Wage (standard div.)	210	21,898 – 94,822	75,813	4,987 – 12,208	9,072
BWWC Mean Average	28	1,646 – 76,440	49,966	503 – 19,682	13,408
Total	3,098				

D Zero-Knowledge Range Proofs

Note on ZK Range Proofs. A range proof [8] is a computational proof that a value lies within a given range. Range proofs can be performed in zero-knowledge protocols, meaning that the verifier learns nothing of the value being tested, only the result of the computation. Such zero-knowledge proofs are useful in anonymous payment systems which require aggregate payment validation without betraying information about individual payments to the broader payment network [34]. These zero-knowledge proofs can be considered a 2PC between a prover (holding a secret value) and a verifier (holding a range to be tested against the value) wherein the functionality executed is the aforementioned range validation check and the privacy of the prover's value must be maintained. The logic of a range proof effectively matches that of the Millionaires problem, and so this target is included simply to highlight McFIL's capacity to analyze a ZK protocol. We offer our software implementation as a tool to researchers and developers of novel ZK protocols as a method to evaluate privacy loss.