ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State

Jeongseok Son Griffin Prechter Rishabh Poddar Raluca Ada Popa Koushik Sen
University of California, Berkeley

Abstract

Encryption of secret data prevents an adversary from learning sensitive information by observing the transferred data. Even though the data itself is encrypted, however, an attacker can watch which locations of the memory, disk, and network are accessed and infer a significant amount of secret information.

To defend against attacks based on this access pattern leakage, a number of oblivious algorithms have been devised. These algorithms transform the access pattern in a way that the access sequences are independent of the secret input data. Since oblivious algorithms tend to be slow, a go-to optimization for algorithm designers is to leverage *space unobservable to the attacker*. However, one can easily miss a subtle detail and violate the oblivious property in the process of doing so.

In this paper, we propose ObliCheck, a checker verifying whether a given algorithm is indeed oblivious. In contrast to existing checkers, ObliCheck distinguishes observable and unobservable state of an algorithm. It employs symbolic execution to check whether all execution paths exhibit the same observable behavior. To achieve accuracy and efficiency, ObliCheck introduces two key techniques: Optimistic State Merging to quickly check if the algorithm is oblivious, and Iterative State Unmerging to iteratively refine its judgment if the algorithm is reported as not oblivious. ObliCheck achieves ×50300 of performance improvement over conventional symbolic execution without sacrificing accuracy.

1 Introduction

Security and privacy have become crucial requirements in the modern computing era. To preserve the secrecy of sensitive data, data encryption is now widely adopted and prevents an adversary from learning secret information by observing the data content. However, attackers can still infer secret information by observing access patterns to the data. Even though the data itself is encrypted, an attacker can watch which locations of the memory, disk, and network are accessed. Such concerns are growing with the increasing adoption of hardware enclaves such as Intel SGX [49], which provides memory encryption but does not hide accesses to memory. By simply observing the access patterns, several research efforts [23, 37, 42, 43, 47, 57, 58, 71] have shown that an attacker can reconstruct secret information such as confidential search keywords, entire sensitive documents, or secret images.

As a result, a rich line of work designs *oblivious* execution to prevent such side channels based on access patterns. There are two types of oblivious algorithms. The first, Oblivious RAM (ORAM) [31, 66], can be used *generically* to hide accesses to memory, and fits best for workloads of the type "point queries". Intuitively, ORAM randomizes accesses to memory. However, even the fastest ORAM scheme incurs polylogarithmic overhead proportional to the memory size per access, which becomes prohibitively slow for processing a large amount of data as in data analytics and machine learning. For these workloads, instead, researchers have proposed a large array of *specialized oblivious* algorithms, such as algorithms for joins, filters, aggregates [7, 11, 14, 19, 57, 78], and machine learning algorithms [36, 48, 58, 64]. These specialized algorithms work by accessing memory according to a *predefined schedule of accesses*, which depends only on an upper bound on the data size and not on data content. In this paper, we focus on such specialized oblivious algorithms.

Oblivious algorithms in general tend to be notoriously slow (e.g., hundreds of times for data analytics [78] and tens of times for point queries [66]). To reduce such overhead, many oblivious algorithms take advantage of an effective design strategy: they leverage *special regions of memory* that are *not observable* to the attacker. Such unobservable memory, albeit often smaller than the observable one, allows the algorithm to make direct and fast accesses to data. It essentially works as a cache for the slower observable memory, which is accessed obliviously. Different techniques choose different resources as unobservable. For example, some techniques [7, 51, 58, 60] treat registers as unobservable but all the cache and main memory as observable in the context of hardware enclaves such as Intel SGX. GhostRider [46] employs an on-chip scratchpad as an unobservable space to make the memory trace oblivious. Certain techniques focus on the network as being observable by an attacker and the internal secure region of a machine as unobservable [57, 78]. These techniques show one or more orders of magnitude [78] performance improvement by leveraging the unobservable memory.

While generic algorithms like ORAM are heavily scrutinized, specialized algorithms designed for different settings do not receive the same level of scrutiny. Further, these algorithms can be quite complex, balancing rich computations with efficiency. The designer can miss a subtle detail and violate the oblivious property. Currently, an oblivious algorithm comes with written proof, and users must verify the proof manually. As a result, recent research efforts devise ways to check whether an algorithm is oblivious in an automated way (by looking for a secret dependent branch) using taint analysis [15, 33, 59, 77]. These techniques, however, cannot discern unobservable state and would classify an algorithm as not oblivious because of its non-oblivious accesses to unobservable state. Thus, they cannot model a vast array of modern oblivious algorithms.

We propose *ObliCheck*, a checker that can verify oblivious algorithms having unobservable state in an efficient and accurate manner. ObliCheck allows algorithm designers to write an oblivious algorithm using ObliCheck's APIs to distinguish between observable and unobservable space. Based on this distinction, ObliCheck precisely records the access patterns visible to an attacker. Then, ObliCheck automatically proves that the algorithm satisfies the obliviousness condition. Otherwise, ObliCheck provides counterexamples – i.e., inputs that violate the oblivious property – and identifies program statements that trigger non-oblivious behavior.

ObliCheck primarily aims to verify the oblivious property of an algorithm, not the actual implementation of the algorithm. We use a subset of JavaScript for modeling algorithms. We made this choice to leverage an existing program analysis framework, Jalangi [61], for ObliCheck's implementation. Moreover, we focus on a subset of the language because verification of programs in the full JavaScript language could result in verification conditions having undecidable theories. Automated verification fails for undecidable theories. We expect that an algorithm designer will use ObliCheck to verify algorithms rapidly during the algorithm design phase, instead of trying to verify the algorithm manually.

1.1 Techniques and contributions

We observed that taint analysis used in prior work [15, 33, 59, 77] is too 'coarse' to capture unobservable state. With taint analysis, if a branch predicate contains tainted variables, then a checker simply rejects the algorithm even if both execution paths of the branch display the same observable behavior. Instead, we observe that we can overcome the limitations of taint analysis with symbolic execution [17, 38]. Using symbolic execution, ObliCheck can analyze an input algorithm with unobservable state in a finer-grained manner and reason about how observable and unobservable state changes in each execution path. Even if a branch depends on a secret input variable, ObliCheck correctly classifies an algorithm as oblivious if the two execution paths after the branch show the same observable behavior. For example, if the two paths both send an identically-sized encrypted message over the network, our checker can conclude both branches maintain the same observable state (the size of the message and its destination) since the message content itself is encrypted (thus unobservable).

However, a naïve application of symbolic execution does not scale. The main challenge with employing symbolic execution is that the program state quickly blows up as the number of branches in the program increases, making it infeasible to complete the check for many algorithms. While traditional state merging [10, 27, 27, 30, 63] can merge states to alleviate the path explosion problem to some extent, it only works when the values in two different paths are the same. To address this problem, ObliCheck employs *a novel optimistic state merging* technique (§4), which leverages the domain-specific knowledge of oblivious algorithms that the actual values are unobservable to the attacker. ObliCheck uses this insight to optimistically merge two different unobservable values by introducing a *new* unconstrained symbolic value for over-approximating the two unobservable values.

Such "aggressive" state merging for symbolic values is effective at tackling path explosion, but could result in a false "not-oblivious" prognosis. If a symbolic variable, x, is merged into an unconstrained new symbolic variable y, later accesses to y in a conditional statement may trigger an execution path which would have been impossible if x were not replaced with unconstrained y. To address this issue, we devise a technique called *iterative state unmerging* (§5). ObliCheck records symbolic variables merged during the execution. Then, it iteratively refines its judgment by backtracking the execution and unmerges a part of merged variables which may have caused the wrong prognosis. This iterative probing process continues until it either classifies the algorithm as oblivious, or completes the refinement process.

Although iterative state unmerging costs extra symbolic execution, we find that the overhead is tolerable. This is because our target algorithms are *mostly* oblivious: an algorithm designer who wants to check their algorithm for obliviousness likely did a decent job making much of the algorithm oblivious, but is worried about subtle mistakes. Hence, most algorithms require few iterations of the iterative state unmerging process, and even when an algorithm needs the extra runs, our evaluation shows that the overhead is less than 70% of single execution time. Further, when ObliCheck reports an algorithm as not oblivious, ObliCheck produces a counterexample that violates the obliviousness verification condition. This information provides valuable help to the algorithm designers to amend their algorithm.

Finally, a well-known limitation of symbolic execution is its inability to verify an algorithm containing an input-dependent loop, requiring the user to provide loop invariants manually, making it hard to verify oblivious algorithms written in terms of an arbitrary length of the input. In ObliCheck, we design a loop summarization technique (§6) that can automatically generate a loop invariant for common loop patterns employed in oblivious algorithms: each iteration of a loop appends the same constant number of elements to the output buffer. Using this observation, ObliCheck can automatically figure out the side-effect of a loop on the output length, enabling it to verify oblivious algorithms not tied to a concrete length of the input.

We evaluated ObliCheck using 13 existing oblivious algorithms, and find that ObliCheck improves the verification performance up to ×50300 over conventional techniques. The checking time of ObliCheck grows linearly as the number of input records grows, whereas that of an existing technique increases exponentially.

2 Background and Existing Approach

We first provide necessary background information regarding the oblivious property and symbolic execution to understand the problems. We then point out the limitations of an existing approach to motivate our approach.

2.1 Oblivious Property and Oblivious Algorithms

The oblivious property implies the access sequences of an algorithm are independent of the secret input data. To achieve the oblivious property in a practical sense, specialized oblivious algorithms have recently been devised. In contrast to Oblivious RAM (ORAM), which compiles a general algorithm and runs it in an oblivious manner, oblivious algorithms are designed for a specific purpose for data processing such as distributed data analytics [57, 78], data structures [22, 32, 70], and machine learning [56, 58]. Instead of randomly shuffling and re-encrypting data as ORAM does, oblivious algorithms implement fixed scheduling independent of secret input data in a deterministic manner.

Oblivious algorithms leverage *unobservable space*, a secure region of registers or memory which an attacker cannot observe. Since the unobservable space is not visible to an attacker, an algorithm can access data inside the unobservable space fast in a non-oblivious way. Existing oblivious algorithms use different types of unobservable space to protect secret data from different types of attackers. For example, oblivious algorithms for distributed data analysis [14, 57, 78] assume a network attacker who can observe network traffic but cannot observe a part of local memory. The network attacker can only watch encrypted messages sent over the network, so the information the attacker can utilize is the network access patterns including the size of the messages and the source and destination network addresses. On the other hand, other works focusing on local data processing [7, 51, 58] regard registers as unobservable space and treat cache and local memory as observable by a memory attacker. We will discuss how ObliCheck captures different threat models under an observable and unobservable space abstraction in §3.1.

2.2 Symbolic Execution and Path Explosion Problem

Symbolic execution runs a program with *symbolic values* as input where symbols represent arbitrary values. During symbolic execution, each feasible execution path of the program is executed symbolically: The execution of each instruction updates the state with symbolic expressions containing the input symbols. The execution of a conditional instruction forks the execution into two separate execution paths—one taking the `true` branch and the other taking the `branch`. Symbolic execution maintains a first-order logic formula, say ϕ, for each path. The execution of a conditional instruction updates the paths conditions along the *then* and *else* paths with $\phi \wedge c$ and $\phi \wedge \neg c$, respectively, where c is the symbolic expression corresponding to the condition in the instruction. At the end of the execution, a constraint solver solves the path condition of each execution path to generate a set of representative inputs that exercise those paths of the program.

One of the most common problems that a user of symbolic execution encounters is path explosion. A traditional symbolic execution forks into two execution paths for each conditional branch. Thus, the number of paths explored and the corresponding state of symbolic values grow exponentially in the number of branches.

2.3 State Merging and MultiSE

One way to alleviate the path explosion problem is state merging [10, 27, 30, 63]. State merging techniques merge the symbolic state of different paths at join points in the control-flow graph to reduce the number of paths to explore. Traditional state merging introduces a new symbolic variable for each merged value. This *auxiliary variable* is used to encode possible distinct values for the same variable in the merged symbolic state. A key issue with traditional state merging is that it could result in constraints that cannot be handled by constraint solvers. MultiSE [63] achieves state merging without auxiliary variables and control-flow analysis. It is based on a new representation of the state called *value summary*. A value summary is a set of guarded symbolic expressions, pairs of a path constraint and a corresponding value of a variable.

For example, after a conditional statement, `if` C `then` $x = x_0$ `else` $x = x_1$, symbolic execution diverges into two paths. The value-summary representation of the state after this statement is $x \mapsto \{(C, x_0), (\neg C, x_1)\}$. This represents the value of x becomes x_0 if the condition C holds, and x_1 otherwise. MultiSE performs state merging incrementally by updating the value-summary of a variable at every assignment statement. MultiSE combines the guarded symbolic expressions with logical disjunction when the values are the same. When $x_0 = x_1$ in the previous case, the merged state is $x \mapsto \{(C \vee \neg C, x_0)\}$, simplified to $x \mapsto \{(True, x_0)\}$.

The benefit of state merging is apparent when the values of a variable on different paths are identical. State merging reduces the execution time by half in this case. When the values are different, however, state merging comes at the cost of com-

Check Result	$Algorithm_0$ is actually:	
	Oblivious	Not Oblivious
$Algorithm_0$ is oblivious	True Negative(✓)	False Negative(✗)
$Algorithm_0$ is not oblivious	False Positive(✗)	True Positive(✓)

Table 1: Definition of the correct and erroneous classification types of an oblivious checker. The null hypothesis is that a given algorithm is oblivious. Rejecting a benign oblivious algorithm is a false positive case (Type I error). Accepting a not oblivious algorithm is a false negative case (Type II error).

```
1  function tag(secretInput, threshold) {
2      var buf = [];
3          for (var i = 0; i < secretInuput.length;
               i++) {
4              if (secretInuput[i] < threshold) {
5                  buf.push(Pair(secretInuput[i], 0));
6              } else {
7                  buf.push(Pair(secretInuput[i], 1));
8              }
9          }
10     }
11     var encrypted = Crypto.encrypt(buf);
12     socket.send(ADDR, encrypted);
13 }
```

Listing 1: An example code from Opaque [3] in Javascript. It tags each element in the secret input and sends the encrypted result over the network. Red variables are tainted variables from the secret input `secretInput[i]`. Since the algorithm has a secret (`secretInput[i]`) dependent branch, taint analysis based techniques deem that this code has leakage although the observed size of the data (`encrypted`) does not depend on the secret input.

plicated path constraints, which increase constraint solving time. In some cases, state merging may lower the performance of symbolic execution if applied indiscreetly [41].

2.4 Existing Approach Using Taint Analysis

Several techniques have been devised to check the access pattern leakage of an algorithm. The most widely used technique is taint analysis. Existing works utilize it to check side-channel leakage [15, 59] and more broadly obliviousness [8, 75]. This line of work identifies variables whose values depend on secret input. They track the taints of variables propagated from secret inputs. In this way, a checker can check whether a given algorithm includes a secret dependent branch. Algorithms with secret dependent branches are rejected in this approach assuming that those branches incur information leakage because of the different behaviors in the true and false blocks of the conditional statements.

Limitation. However, taint analysis can reject benign oblivious programs many times. Even if both execution paths of a branch exhibit the same observable behavior, a checker simply rejects the algorithm if the branch contains a tainted variable. As we define in Table 1, this is a false-positive error. For example, let us assumes the network attacker discussed in §2.1. The attacker can only observe the network access patterns including the size of data sent over the network, but not the actual content of the encrypted data. Listing 1 shows one example algorithm where taint analysis leads to a false positive. In this example, the predicate (Line 4) contains a secret variable `secretInput[i]`. Hence, taint tracking based techniques reject this algorithm due to this secret branch. However, since the threat model in oblivious algorithms assumes the actual content (`(secretInput[i], 0)` in Line 5, `(secretInput[i], 1)` in Line 7) is encrypted, both true and false branch blocks have indistinguishable behavior to an attacker. Hence, the example algorithm is actually oblivious.

Requirements. A more accurate checker for oblivious algorithms should satisfy the following requirements.

1) Be aware of which state of a program is observable or not to an attacker (e.g., in Listing 1, the data content is encrypted, thus invisible, but the size of the data is revealed).
2) Understand the behavior of a program on different execution paths across the whole input space to make a sound judgment of whether an algorithm is oblivious.
3) Know which input values are secret or public to decide the behavior of a program is independent of secret input.
4) Since a checker has a limited time budget, the checking process should be scalable in terms of the number of input data records.

3 ObliCheck Overview

In order to check oblivious algorithms with unobservable state and overcome the limitations of existing approaches, we propose ObliCheck. We now provide an overview of ObliCheck's API, the threat model it assumes, and its security guarantees.

3.1 ObliCheck APIs

To provide a framework that can accommodate algorithms with different threat models, ObliCheck provides abstract observable and unobservable memory space. Any read and write operations to the observable space are assumed to be observed by an attacker. ObliCheck provides algorithm designers with special APIs for describing reads and writes to the observable space as described in Table 2. We assume data written to or read from observable space is always encrypted. Thus, an attacker can learn the size, source/destination address of the data, and the type of operation (read or write) but not the actual content. Using this abstract store model with APIs, a designer can reflect a threat model that she assumes in the code.

ObliCheck offers two categories of APIs for a designer to write an oblivious algorithm. The first has functions that describe communication between unobservable and observable spaces. The second one is to specify whether an input value is secret or public. Table 2 lists the APIs that ObliCheck provides. Using *observableRead* and *observableWrite*, a designer can naturally render a boundary between observable and observable spaces in the algorithm.

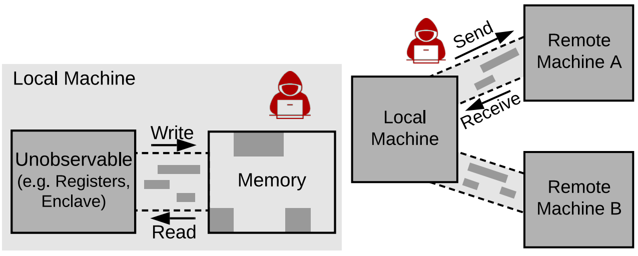
ObliCheck keeps the access sequence under the hood and uses the access sequence to check the final verification condi-

Name	Arguments	Description	Effect
observableWrite	(space, addr, buf)	Write buf at the addr of observable space	τ_P += (<space.ID,W>, addr, size(buf)), space.store[addr] = *buf
observableRead	(space, addr, buf)	Read size(buf) of bytes at addr of observable space	τ_P += (<space.ID,R>, addr, size(buf)), *buf = space.store[addr]
readSecretInput	()	Introduce a secret input	A new tainted symbolic value is added
readPublicInput	()	Introduce a public input	A new untainted symbolic value is added

Table 2: API of ObliCheck. *observableWrite* and *observableRead* are used to describe communication between observable and unobservable space. τ_P is the trace of observations defined as a sequence of triplets in § 3.3. The first field of a triplet added to the access sequence contains the enumerated type of access of MW, MR, NS, and NR, which encode memory write, memory read, network send and network receive respectively. *readSecretInput*, and *readPublicInput* are necessary to make ObliCheck distinguish the secret inputs from public inputs (Refer to Figure 3).

Function	Implementation using ObliCheck API
send(dst, buf)	*observableWrite*(network, <host, dst>, buf)
recv(src, buf)	*observableRead*(network, <src, host>, buf)
write(dst, buf)	*observableWrite*(memory, dst, buf)
read(src, buf)	*observableRead*(memory, src, buf)

Table 3: Example user-defined functions accessing observable spaces. *send* and *recv* are used to express message transfer over network and *read* and *write* represents local memory access. network and memory are initialized by users with unique IDs and memory space to store written and sent data.



(a) Memory Attacker **(b)** Network Attacker

Figure 1: Threat model of ObliCheck. The dark gray (■) part of the figure represents the store and data that an adversary cannot observe. The light gray(□) indicates observable parts. An attacker is not able to eavesdrop on the unobservable space and the content of encrypted data. However, an attacker is capable of learning the size of transferred data, the locations of data written to or read from an observable space, and the destination and source network addresses of the network messages and their sizes.

tion explained in §3.3. *readSecretInput* and *readPublicInput* let a designer specify the secret input of an algorithm. This specification is necessary to generate the verification condition at the end of symbolic execution. Listing 2 shows the code in Listing 1 re-written using ObliCheck's API.

3.2 Threat Model

As discussed in §3.1, ObliCheck assumes the existence of unobservable spaces where an attacker cannot watch the data content and access patterns. ObliCheck considers an attacker that watches any accesses to observable space, as depicted in Figure 1. However, we assume the attacker cannot learn about the actual content of data written to or read from observable space because the data is encrypted when it crosses the boundary between unobservable and observable spaces.

```
function tag(secretInput, threshold) {
    var buf = [];
    for (var i = 0; i < secretInput.length; i++) {
        if (secretInput[i] < threshold) {
            buf.push(Pair(secretInput[i], 0));
        } else {
            buf.push(Pair(secretInput[i], 1));
        }
    }
    send(ADDR, buf);
}
function main(n) {
    var secretInput = new Array(n);
    for (var i = 0; i < n; i++) { secretInput[i] =
        ObliCheck.readSecretInput() };
    var threshold = ObliCheck.readPublicInput();
    tag(secretInput, threshold);
}
```

Listing 2: Listing 1 is re-written using the APIs of ObliCheck. Only the socket.send is replaced with send, and the input is introduced using readSecretInput, and readPublicInput.

It is important to note that this observable space can differ between threat models. A memory attacker in Figure 1a can observe the memory address and size of data written to and read from memory. A network attacker in Figure 1b can watch the network address and length of messages transferred. To account for this variability of unobservable and observable spaces, ObliCheck provides an abstract threat model.

This abstract threat model allows algorithm designers to express common threat models that oblivious algorithms assume using the APIs of ObliCheck. For example, the network attacker discussed in §1 can be modeled by using *observableWrite* and *observableRead* for network *send* and *receive* functions respectively. The memory attacker can be modeled similarly. Table 3 shows how these functions can be defined using APIs of ObliCheck. We focus on the network adversary as a running example, but an algorithm assuming the memory attacker can be checked the same way.

ObliCheck only checks the obliviousness of a given algorithm and assumes the data is properly encrypted when it is written to an observable location. Mistakes of not properly encrypting data can be caught using existing information flow checking techniques [20, 26, 35, 40, 53, 55, 72–74].

3.3 Security Guarantee

To formulate the security guarantees of ObliCheck, we first define the *trace of observations* visible to the adversary during an execution. Given an algorithm P with input I, the trace of observations τ is defined as a sequence of triplets:

$$\tau_P(I) = < (t_i, a_i, l_i) | i \in N >$$

where t represents a type of access, a denotes a target or source location of the operation, and l represents the size of a data read or written. The type of access is either read or write combined with the type of an observable space (*e.g.*, memory or network). Further, since we assume the data itself is encrypted properly before being written to an observable store, the attacker can only observe the size of the data that is read or written, and not the actual contents.

Note that in addition to secret data, an algorithm P may also receive some public data as input. For P to achieve the oblivious property, we require that given any pair of inputs I and I', as long as the public input is the same, then no polynomial-time adversary should be able to distinguish between the traces $\tau_P(I)$ and $\tau_P(I')$. Based on this definition, a condition for checking the oblivious property can be expressed as follows:

$$\begin{aligned} &\forall I,\ I' \in InputSpace(P), \\ &\quad PublicInput_P(I) = PublicInput_P(I') \\ &\qquad \Rightarrow \tau_P(I) = \tau_P(I') \end{aligned}$$

Here, *InputSpace* represents all the possible input spaces of a given algorithm, and $PublicInput_P$ returns the public input of an algorithm P. ObliCheck verifies that the above condition holds while checking an algorithm. The condition assumes nothing about *SecretInput*, which encodes the independence of the observable output from secret input.

ObliCheck records the trace during the execution under the hood when it encounters a read or write API explained in §3.1. The verification condition is written in terms of the pairs of input (I, I'). This implies that the verification condition for the oblivious property is a 2-safety property [67] that requires a checker to observe two finite traces of an algorithm. We will describe how ObliCheck uses symbolic execution to check the above verification condition in §4.1.

4 Symbolic Execution and State Merging

4.1 Symbolic Execution for Checking Obliviousness

ObliCheck executes an algorithm symbolically, and at the end of the execution, it checks whether the algorithm satisfies the obliviousness condition defined in §3.3. ObliCheck uses symbolic execution in the following way.

ObliCheck starts by treating all input values as symbolic variables. ObliCheck explores both the true and false blocks of all branches containing a symbolic value, while distinguishing between secret and public symbolic variables to correctly generate the verification condition at the end of the execution.

However, just running an algorithm once symbolically is not sufficient because the verification condition of obliviousness is written in terms of pairs of input. In other words, obliviousness is a 2-safety property. Terauchi and Aiken [67] formally defined a 2-safety property to distinguish it from a general safety property, which can be proved by observing a single finite trace.

In order to refute a 2-safety property, a checker has to observe two finite traces of an algorithm. Hence, ObliCheck internally runs the algorithm twice symbolically, by sequentially composing two copies of the algorithm. Each execution path of the first copy is followed by each one of the second copy. This makes ObliCheck explore every pair (Cartesian product) of the execution paths with pairs of input $(I, I') \in InputSpace(P)$. At the end of the second execution, ObliCheck compares the traces of both runs and checks that the verification condition is always true using a constraint solver (which checks that the negation of the verification condition is unsatisfiable).

Example. To demonstrate how symbolic execution is used, we represent the value-summary symbolic state of Listing 2 in Table 4. For brevity, we assume the input length `n` is 1 so the loop iterates only once and omit the program counter (*pc*) state. We will generalize for algorithms with loops bounded by an arbitrary symbolic value in §6.

Line	Value Summary
2-4	`buf.length` $\mapsto \{(true, 0)\}$, `i` $\mapsto \{(true, 0)\}$, `buf[i]` $\mapsto \{(true, undefined)\}$
5,8-10	`buf.length` $\mapsto \{(x_{0,first} < y_{first}, 1)\}$, `i` $\mapsto \{(x_{0,first} < y_{first}, 0)\}$, `buf[i]` $\mapsto \{(x_{0,first} < y_{first}, Pair(x_{0,first}, 0))\}$
7,8-10	`buf.length` $\mapsto \{(x_{0,first} \geq y_{first}, 1)\}$, `i` $\mapsto \{(x_{0,first} \geq y_{first}, 0)\}$, `buf[i]` $\mapsto \{(x_{0,first} \geq y_{first}, Pair(x_{0,first}, 1))\}$
2-4	`buf.length` $\mapsto \{(true, 0)\}$, `i` $\mapsto \{(true, 0)\}$, `buf[i]` $\mapsto \{(true, undefined)\}$
5,8-10	`buf.length` $\mapsto \{(x_{0,second} < y_{second}, 1)\}$, `i` $\mapsto \{(x_{0,second} < y_{second}, 0)\}$, `buf[i]` $\mapsto \{(x_{0,second} < y_{second}, Pair(x_{0,second}, 0))\}$
7,8-10	`buf.length` $\mapsto \{(x_{0,second} \geq y_{second}, 1)\}$, `i` $\mapsto \{(x_{0,second} \geq y_{second}, 0)\}$, `buf[i]` $\mapsto \{(x_{0,second} \geq y_{second}, Pair(x_{0,second}, 1))\}$

Table 4: Result of symbolic execution of the algorithm in Listing 2.

`main` introduces secret and public symbolic variables x_0 and y respectively and assigns them to `secretInput[0]` and `threshold`. To differentiate the first and second symbolic executions, we add additional subscripts $first$ and $second$ to the variables. Inside the `tag` function, the first symbolic execution starts with an initial path condition $True$ and the length of the output buffer is 0. After encountering the branch at Line 4, the execution diverges into two sets and the output buffer length increments by one. The second symbolic execution runs the same algorithm but with different symbolic variables: $x_{0,second}$ and y_{second} instead of $x_{0,first}$ and y_{first}.

After finishing the symbolic execution, ObliCheck generates a verification condition based on the definition in §3.3:

$$
\begin{aligned}
& y_{first} = y_{second} \Rightarrow \\
& \quad ((x_{0,first} < y_{first} \land x_{0,second} < y_{second} \Rightarrow 1 = 1) \\
& \quad \land (x_{0,first} < y_{first} \land x_{0,second} \geq y_{second} \Rightarrow 1 = 1) \\
& \quad \land (x_{0,first} \geq y_{first} \land x_{0,second} < y_{second} \Rightarrow 1 = 1) \\
& \quad \land (x_{0,first} \geq y_{first} \land x_{0,second} \geq y_{second} \Rightarrow 1 = 1))
\end{aligned}
$$

This formula is trivially always true since `buf.length` is always a concrete value 1 (we leave out the type of access and the address fields of the trace for simplicity). The verification condition is quite trivial for this simple example, but as an input algorithm becomes more complicated, symbolic execution proves its real worth since it can capture how the observable trace changes over the execution and can exercise all possible execution paths.

4.2 Optimistic State Merging

As we discussed in § 2.3, existing state merging techniques merge states on different paths to alleviate the path explosion problem. When a variable carries distinct values along different paths, however, the benefit of state merging diminishes. In MultiSE, for example, the size of value summary can still grow exponentially if the variable maintains different values across all execution paths. To solve this problem, we devise *optimistic state merging* – a state merging technique that leverages domain-specific knowledge of oblivious execution in the presence of unobservable state.

Shortcomings of Traditional State Merging. In Listing 1, the code is oblivious under the definition in §3.3 assuming the data length is public. The algorithm always sends the buffer with a length `n` regardless of the secret values in `secretInputRecords`. To check this condition, a checker should confirm the length of `encrypted` is the same across any possible pairs of `secretInputRecords`. Naïvely running symbolic execution leads to path explosion because the branch is inside the `for` loop. Since it is common to iterate over elements in the input data set within unobservable space, we need a way to prevent path explosion in this case.

To mitigate the path explosion problem, state merging techniques merge two different symbolic states of a variable. As we discussed in § 2.3, this can prevent unnecessary exploration. However, conventional state merging techniques do not effectively reduce the paths to explore when two merged states are different from each other. For example, Table 4 shows the symbolic states after the execution in Listing 2. With traditional state merging, the `true` and `false` paths of the `if` statement at Line 4 cannot get combined because `buf[i]` has different state in each path. In other words, traditional state merging techniques are sound and complete with regard to symbolic execution and explore the same set of program behaviors as regular symbolic execution.

Merging Paths Using Domain Specific Knowledge of Oblivious Algorithms. ObliCheck is able to apply state merging more aggressively through a domain specific insight. Optimistic state merging leverages the observation that, in oblivious algorithms, *the attacker is unable to distinguish between different unobservable states because the plaintext data only resides in unobservable space*, and is later encrypted when written to observable space. For example, `buf[i]` in Listing 2 is encrypted when the `buf` is sent over network at Line 10. Therefore, at branching statements, ObliCheck explores both true and false blocks immediately and merges the corresponding states into a new symbolic variable without divergence.

ObliCheck simplifies path conditions by introducing a new variable when merging two different symbolic expressions. For example, the algorithm in Listing 2 exhibits different state of `buf[i]` in the `then` and `else` branches after Line 4 (Pair(x_0,0) and Pair(x_0,1) respectively; Table 4). Hence, traditional state merging cannot merge these two states. In contrast, ObliCheck introduces a new unconstrained symbolic variable, z. Now, `buf[i][1]` becomes the same z, so those two states can get combined as in Table 5.

Line	Value Summary
2-4	$\texttt{buf.length} \mapsto \{(true,0)\}, \texttt{i} \mapsto \{(true,0)\}, \texttt{buf[i]} \mapsto \{(true, undefined)\}$
5	$\texttt{buf.length} \mapsto \{(x_{0,first} < y_{first}, 1)\}, \texttt{i} \mapsto \{(x_{0,first} < y_{first}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,first} < y_{first}, Pair(x_{0,first}, 0))\}$
7	$\texttt{buf.length} \mapsto \{(x_{0,first} \geq y_{first}, 1)\}, \texttt{i} \mapsto \{(x_{0,first} \geq y_{first}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,first} \geq y_{first}, Pair(x_{0,first}, 1))\}$
8-10	$\texttt{buf.length} \mapsto \{(true,1)\}, \texttt{i} \mapsto \{(true,0)\}, \texttt{buf[i]} \mapsto \{(true, Pair(x_{0,first}, z))\}$
2-4	$\texttt{buf.length} \mapsto \{(true,0)\}, \texttt{i} \mapsto \{(true,0)\}, \texttt{buf[i]} \mapsto \{(true, undefined)\}$
5	$\texttt{buf.length} \mapsto \{(x_{0,second} < y_{second}, 1)\}, \texttt{i} \mapsto \{(x_{0,second} < y_{second}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,second} < y_{second}, Pair(x_{0,second}, 0))\}$
7	$\texttt{buf.length} \mapsto \{(x_{0,second} \geq y_{second}, 1)\}, \texttt{i} \mapsto \{(x_{0,second} \geq y_{second}, 0)\},$ $\texttt{buf[i]} \mapsto \{(x_{0,second} \geq y_{second}, Pair(x_{0,second}, 1))\}$
8-10	$\texttt{buf.length} \mapsto \{(true,1)\}, \texttt{i} \mapsto \{(true,0)\}, \texttt{buf[i]} \mapsto \{(true, Pair(x_{0,second}, z))\}$

Table 5: Result of optimistic state merging of the Listing 2.

This merging simplifies the verification condition to $y_{first} = y_{second} \Rightarrow 1 = 1$, which reduces the burden of a constraint solver. Optimistic state merging is an over-approximation based on the domain-specific knowledge of oblivious algorithms, where the data is encrypted and not observable by an adversary. Since it is an over-approximation, this a sound transformation; namely, if the transformed symbolic execution judges an algorithm is oblivious, then the original algorithm is always oblivious.

Tracking the Secret Values after Merging. ObliCheck checks the verification after the execution of two copies of a given algorithm. The verification condition in §3.3 is generated from the access sequence recorded by ObliCheck under the hood. To generate the verification condition, ObliCheck needs to know which symbolic values are secret or public.

To this end, ObliCheck associates a taint tag with every introduced symbolic variable. Symbolic variables introduced by *readSecretInput* are assigned a taint tag 1, and the others are assigned 0. ObliCheck sees the taint tag of symbolic values included in the trace and produces a proper verification condition based on this information. Figure 3 describes the semantics in a formal notation.

The use of taint tags is necessary due to optimistic state

Pgm	::=	$(\ell\colon\ stmt\,;)^*$
stmt	::=	$x = c$
		$x =\ readSecretInput$
		$x =\ readPublicInput$
		$z = x \bowtie y$
		if x goto y
		$y = *x$
		$*x = y$
		error
		halt
where		
Σ		is the program state
V		is a set of variables
C		is the set of constants
L		is the set of statement labels
A		is a set of memory addresses
x, y, z		are elements of V
pc		an element of V denoting the program counter
c		is an element of $C \cup A \cup L$
ℓ		is an element of L
$\bowtie$		is a binary operator
SecretSet		is a set of secret symbolic variables
PublicSet		is a set of public symbolic variables

Figure 2: A simple imperative language originally devised by Sen *et al.* in MultiSE [63], augmented with states *SecretSet* and *PublicSet* to maintain the mapping from symbolic values to the taint state. The functions *readSecretInput* and *readPublicInput* introduce a symbolic variable and initialize the corresponding taint tag. Refer to Figure 3 for more details.

merging. When ObliCheck applies optimistic state merging, it has to maintain whether a newly generated symbolic variable is secret. Taint tags let ObliCheck track how secret input is propagated and decide the security level of a newly generated symbolic variable after optimistic state merging. Unlike traditional taint analysis, ObliCheck draws the final verdict based on the verification condition, not the value of taint tags.

Optimistic State Merging Semantics. Our optimistic state merging technique is based on MultiSE [63]. MultiSE merges state without introducing auxiliary variables, and does not require control flow graph analysis to identify join points because the merging is done incrementally per assignment operation. MultiSE maintains the state of variables in the form of a value summary – a set of path conditions and possible values of a variable. Each pair represents a possible value which a variable can have and the corresponding condition that leads to it. For example, `buf.length` in Listing 2 can be represented using value summary $\{(x_0 < y, 1), (x_0 \geq y, 1)\}$ after the first loop iteration.

In MultiSE, state merging can be done by simply replacing pairs with the same values with a single pair whose path condition is the disjunction of the conditions of the merged pairs. For instance, the value summary of `buf.length`, $\{(x_0 < y, 1), (x_0 \geq y, 1)\}$, becomes $\{(True, 1)\}$ after state merging. MultiSE further removes pairs whose path condition is false when merging.

To formally demonstrate the semantics of ObliCheck operations including optimistic state merging, we bring a simple imperative language from MultiSE [63] in Figure 2. Figure 3 defines the operational semantics of ObliCheck. Each operator updates the program state Σ. The initial state maps each variable to $\{(True, \perp)\}$, and pc to $\{(True, l_0)\}$. To incorporate the taint tag, we extend the value part of the value summary from (ϕ, v) to $(\phi, \langle v, t\rangle)$, where t is the taint tag either T or F associated with the value. $\uplus$ is the original value-summary union operator that performs state merging in MultiSE. To distinguish our optimistic state merging operator from the MultiSE operator, we introduce the $\mathbin{\cup\mkern-12mu{\scriptstyle\times}}$ operator in the semantics description. Our optimistic state merging operator works as follows.

- In the value-summary pairs, the value part has an additional taint tag t. T denotes that the corresponding value is secret, and F denotes the value is public.
- For any two pairs $(\phi, \langle v, t\rangle)$ and $(\phi', \langle v', t'\rangle)$ where $v = v'$, a new value summary for s is calculated in the same way as $\uplus$ does except that the new taint tag is set to $t \vee t'$. The new value summary becomes $(s \backslash \{(\phi, \langle v, t\rangle), (\phi', \langle v', t'\rangle)\}) \cup \{(\phi \vee \phi', \langle v, t \vee t'\rangle)\}$.
- For any two pairs (ϕ, v) and (ϕ', v') where $v \neq v'$ in a value summary for s, a new symbolic variable y is introduced. If ϕ or ϕ' contain a secret symbolic variable, the new value summary becomes $(s \backslash \{(\phi, \langle v, t\rangle), (\phi', \langle v', t'\rangle)\}) \cup \{(\phi \vee \phi', \langle y, T\rangle)\}$. Otherwise, the value summary becomes $(s \backslash \{(\phi, \langle v, t\rangle), (\phi', \langle v', t'\rangle)\}) \cup \{(\phi \vee \phi', \langle y, t \vee t'\rangle)\}$

For example, `buf[i]` in Listing 2 has a value summary $\{(x_0 < y, \langle 0, F\rangle), (x_0 \geq y, \langle 1, F\rangle)\}$. After merging, the new value summary becomes $\{(True, \langle z, T\rangle)\}$. The taint tag after merging is T because the original path conditions contain x_0, a secret symbolic variable even though the original merged values 0 and 1 are not secret values.

The $\mathbin{\cup\mkern-12mu{\scriptstyle\times}}$ operator is used in Figure 3 to describe the semantics of symbolic execution and merging techniques used by ObliCheck. Note that the program counter is treated in the same way as MultiSE using $\uplus$ operator.

5 Iterative State Unmerging

Although our optimistic state merging technique improves the performance of ObliCheck without losing soundness, the overapproximation of the technique incurs false positives. In this section, we point out the problem of optimistic state merging and devise a technique that iteratively and selectively removes false positives.

5.1 Problem of Aggressive State Merging

Optimistic state merging overapproximates the values to get merged. This overapproximation enables more values to be merged but loses path-specific information. Because the values are replaced with symbolic variables which can be an arbitrary value satisfying a corresponding path condition, it brings up more false positives.

DOMAIN SPECIFIC GUARDED UPDATE

$$\frac{}{\{(\phi_i^a, < v_i^a, t_i^a >)\}_i \uplus_{\phi} \{(\phi_j^b, < v_j^b, t_j^b >)\}_j = \{(\neg\phi \wedge \phi_i^a, < v_i^a, t_i^a >)\}_i \uplus \{(\phi \wedge \phi_j^b, < v_j^b, t_j^b >)\}_j}$$

NEXTPC

$$\frac{}{NextPC(\Sigma, \phi, \ell) = (\Sigma(\ pc\ ) \setminus \{(\phi, \ell)\}) \uplus \{(\phi, \ell+1)\}}$$

CONSTANT

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (x = c)}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_{\phi} \{(true, < c, F >)\}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

SYMBOLIC PUBLIC INPUT

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (x = \ readPublicInput\ ) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_{\phi} \{(true, < s, F >)\}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)][PublicSet \mapsto \Sigma(PublicSet) \cup \{s\}]}$$

SYMBOLIC SECRET INPUT

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (x = \ readSecretInput\ ) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_{\phi} \{(true, < s, T >)\}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)][SecretSet \mapsto \Sigma(SecretSet) \cup \{s\}]}$$

BINARY OPERATION

$$\frac{\begin{array}{c}(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (z = x \bowtie y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, t_i^x >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, < v_j^y, t_j^y >)\}_j \\ \phi_{ij}^{x \bowtie y} = \phi_i^x \wedge \phi_j^y \qquad v_{ij}^{x \bowtie y} = v_i^x \bowtie v_j^y \qquad t_{ij}^{x \bowtie y} = t_i^x \vee t_j^y\end{array}}{\Sigma \longrightarrow \Sigma[z \mapsto \Sigma(z) \uplus_{\phi} \{(\phi_{ij}^{x \bowtie y}, < v_{ij}^{x \bowtie y}, t_{ij}^{x \bowtie y} >)\}_{ij}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

CONDITIONAL

$$\frac{\begin{array}{c}(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (\text{ if } x \text{ goto } y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, \ell_j^y)\}_j \\ s = \{(\phi_i^x \wedge v_i^x \wedge \phi_j^y, \ell_j^y)\}_{ij} \uplus \{((\phi_i^x \wedge \neg v_i^x), \ell + 1)\}_i\end{array}}{\Sigma \longrightarrow \Sigma[\ pc\ \mapsto (\Sigma(\ pc\ ) \setminus \{(\phi, \ell)\}) \uplus_{\phi} s}$$

LOAD

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (y = *x) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(v_i^x) = \{(\phi_{ij}, < v_{ij}, t_{ij} >)\}_j}{\Sigma \longrightarrow \Sigma[y \mapsto \Sigma(y) \uplus_{\phi} \{(\phi_i^x \wedge \phi_{ij}, < v_{ij}, t_{ij} >)\}_{ij}][\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

STORE

$$\frac{(\phi, \ell) \in \Sigma(\ pc\ ) \qquad Pgm(\ell) = (*x = y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, < v_j^y, t_j^y >)\}_j}{\Sigma \longrightarrow \Sigma[v_i^x \mapsto \Sigma(v_i^x) \uplus_{\phi \wedge \phi_i^x} \{(\phi_j^y, < v_j^y, t_j^y >)\}_j)]_i[\ pc\ \mapsto NextPC(\Sigma, \phi, \ell)]}$$

Figure 3: The semantics of symbolic execution and state merging techniques of ObliCheck. The semantics incorporates the taint tag into the MultiSE semantics [63] in order to track the propagation of secret input through merged symbolic values.

Listing 3 is a benign oblivious algorithm but is reported as not oblivious if our optimistic state merging is used. At Lines 6 and 8, the $i-th$ position of buf is updated to either 0 or 1 depending on the value of secretInput[i]. Since $0 \neq 1$, our optimistic state merging operation introduces a new symbolic variable and puts it in the value summary of buf[i].second. At Lines 16 and 18, the predicates in the branches contain record.second, where each record points to the value stored at buf[i]. Since ObliCheck overapproximated buf[i].second, it has no way to know 0 and 1 are the only possible values for record.second and thus the algorithm is reported as not oblivious.

Our merging technique does not affect the soundness of ObliCheck, but sacrifices the completeness due to the overapproximation for merging. In fact, if we merge every variable, any algorithm that has a secret dependent branch that affects the access sequence is classified as not oblivious, the same way as a taint analysis based checker does. For better precision, ObliCheck has to intelligently choose variables to apply the optimistic state merging technique.

5.2 Iteratively and Selectively Unmerging State

To overcome the issue, we introduce an iterative way to remove false positives. Choosing which values to merge during the execution is tricky. The symbolic execution engine does not immediately know how an updated variable is used later by the verification condition. A naïve solution is rolling back the merged state after the first iteration. However, this simple delayed rollback approach can cause the performance to significantly deteriorate when a given algorithm is a false-positive. In this strawman solution, ObliCheck will always unmerge every symbolic value in the second iteration and perform as poorly as regular symbolic execution.

Instead of identifying which variables to merge, ObliCheck does the reverse. ObliCheck first runs a program merging every variable updated in multiple execution paths. Then it checks the verification condition, and identifies which variables should be unmerged. In the next iteration, ObliCheck backtracks the execution, locates operations where the merging should be avoided and re-runs the program symbolically. The verification is performed again at the end of the iteration. This iterative process helps ObliCheck learn how a certain

```
function tag(secretInput, threshold) {
    var buf = [];
    for (var i = 0; i < secretInput.length; i++) {
        if (secretInput[i] < threshold)
            buf.push(Pair(secretInput[i], 0));
        else
            buf.push(Pair(secretInput[i], 1));
    }
    return buf;
}
function apply(records, func0, func1) {
    var buf = [];
    for (var i = 0; i < records.length; i++) {
        if (records[i].second == 0)
            buf.push(func0(records[i].first));
        if (records[i].second == 1)
            buf.push(func1(records[i].first));
    }
    return buf;
}
function main() {
    // Input values are initialized
    ...
    var tagged = tag(secretInput, publicThreshold);
    ...
    var applied = apply(tagged, funcA, funcB);
    ...
    applied = Cipher.encrypt(applied);
    write(ADDR, applied);
}
```

Listing 3: Tag&Apply code from Opaque [78]. `tag` function tags 0 or 1 depending on the value of each `secretInput[i]`. `apply` applies a function to the value depending on the tag of an element. Optimistic state merging merges the tags 0 and 1 into a symbolic value. Although the branches in `apply` do not cause non-oblivious behavior, the algorithm is reported as non-oblivious because the `record.second` becomes a symbolic value after merging.

merging operation affects the outcome of verification later.

Algorithm 1 in Figure 4 is a formal description of the iterative state unmerging process. During the execution, ObliCheck tracks the location of operations which incur the domain-specific merging. Jalangi inserts a unique operation ID for every operation in a program statically. ObliCheck stores the ID of operations which introduce a symbolic variable or triggered domain-specific merging to an introduced symbolic variable. At the end of each iteration, symbolic variables included in the verification condition are extracted. If the verification condition does not hold and the extracted symbolic variables contain ones introduced by domain-specific merging, the operation IDs stored in *SymVarToOID* are added to *UnmergeOID* to prohibit merging at these locations in the next iteration. This iterative process enables an efficient selection of merging points that do not incur false positive errors.

An algorithm with more non-oblivious branches will end up enduring more unnecessary iterations, wasting time. However, our domain-specific merging was based on the expectation that developers checking an algorithm for obliviousness likely put effort towards making it oblivious, while potentially missing a few details. Therefore, the number of iterations required to unmerge relevant symbolic values is not large. In §7, we evaluate the additional cost using example algorithms. If ObliCheck fails to check an algorithm within a given time budget, it reports the locations where state merging has happened. This information can greatly assist an algorithm designer to manually inspect only a part of the code and then figure out whether the algorithm is a true-positive or false-positive.

Algorithm 1 Iterative state unmerging algorithm

```
global variables
    SymVarToOID                ▷ Symbolic variables to operation IDs
    UnmergeOID                 ▷ Set of operation IDs
end global variables
▷ Called for every assignment operation in a program
procedure UPDATE(OperationID)
    if OperationID $\in$ UnmergeOID then
        CONVENTIONALMERGING(OperationID)
    else
        s $\leftarrow$ DOMAINSPECIFICMERGING(OperationID)
        SymVarToOID[s] $\leftarrow$ SymVarToOID[s] $\cup$
                {OperationID}
procedure OBLICHECKMAIN(Program)
    while true do
        Reset SymVarToOID
        Trace1 $\leftarrow$ SYMBOLICEXEC(Program)
        Trace2 $\leftarrow$ SYMBOLICEXEC(Program)
        VC $\leftarrow$ OBLIVIOUSVC(Trace1, Trace2)
        if VC then
            report OBLIVIOUS, return
        SymVarsInVC $\leftarrow$ EXTRACTSYMVARS(VC)
        if SymVarsInVC $\cap$ SymVarToOID.keys $= \varnothing$ then
            report NOT OBLIVIOUS, return
        for all s $\in$ SymVarsInVC do
            UnmergeOID $\leftarrow$ UnmergeOID $\cup$
                    SymVarToOID[s]
```

Figure 4: A formal description of how our iterative state unmerging algorithm functions. *SymVarToOID* is a dictionary maps a symbolic variable introduced by merging to a set of operation IDs. The operation IDs uniquely identify each operation in a program statically. *UnmergeOID* is a set of operation IDs that represent the locations where ObliCheck should avoid performing our domain-specific merging. For every iteration, *UnmergeOIDs* grows. This lets ObliCheck increases the precision gradually as necessary.

6 Handling Input-dependent Loops

6.1 Limitation of Symbolic Execution: Handling Loops Bounded by Symbolic Expression

A well-known limitation of symbolic execution is its inability of verifying a program containing an input-dependent loop. These types of loops are bounded by a symbolic expression which consists of symbolic input variables. A program with an input-dependent loop has an infinite number of paths to explore. For example, Listing 4 shows a loop bounded by

```
// threshold and inputSize are public input
function tag(secretInput, threshold, inputSize) {
    var buf = [], i = 0;
    while (i < inputSize) {
        if (secretInput[i] < threshold) {
            // buf.length += 1 inside push
            buf.push(Pair(secretInput[i], 0));
        } else {
            // buf.length += 1 inside push
            buf.push(Pair(secretInput[i], 1));
        }
        i++;
    }
    return buf;
}
```

Listing 4: `tag` function with an input-dependent loop. The `for` loop is transformed into `while` to better demonstrate the control flow.

`inputSize`. The path condition of the first iteration inside the loop is $0 <$ `inputSize`. That of the second one is $\neg(0 <$ `inputSize`$) \wedge (1 <$ `inputSize`$)$ and a new path condition is generated infinitely since `inputSize` is not bounded.

Most oblivious algorithms involve loops bounded by symbolic input variables. These loops are used to iterate over an secret input record of which the length is public. The length of the processed output is thus dependent on the input length. However, the algorithm can still be oblivious since revealing the input length does not violate the obliviousness property. In order to verify generalized oblivious algorithms with symbolic input length, ObliCheck is required to handle loops bounded by symbolic variables.

6.2 Automatic Generation of Loop Invariants

In a general program verification, a user is required to provide a loop invariant manually since it is an undecidable problem [24, 34, 44, 65]. However, ObliCheck automatically infers relevant partial loop invariants by leveraging a fact that the length of the output is an *induction variable*. Induction variables get incremented or decremented by a fixed amount for each iteration in a loop. Oblivious algorithms use input-dependent loops to build up output data by iterating over the secret input records. To preserve obliviousness, a fixed amount of elements are appended to the output buffer for every iteration as shown in the tagging example of Listing 4.

As long as the size of a buffer is an induction variable, the problem is reduced to inferring the number of iterations of a loop. The side-effects of a loop to induction variables can be captured by multiplying the delta of the variables per iteration by the number of iterations. Godefroid and Luchaup [29] formalized this idea in dynamic test generation. We extend the idea to capture partial loop invariants in pure symbolic execution. In a similar way that Godefroid and Luchaup [29] proposed, ObliCheck tracks the modified variables and check the delta of the variables and expression in the loop condition between two consecutive iterations. Unlike Godefroid and Luchaup, however, we use pure symbolic execution for sound verification and finish loop summarization within three iterations by over-approximation. Algorithm 2 in Figure 5 describes our loop summarization algorithm.

Algorithm 2 Automatic loop invariant generation algorithm

```
▷ Called for every read operation in a loop
procedure READLOOP(L, Var)
    if Var not in L.UpdatedVars.Keys then
        L.UpdatedVars[Var] = readSecretInput
    return L.UpdatedVars[Var]
▷ Called for every write operation in a loop
procedure UPDATELOOP(L, Var, Val)
    L.UpdatedVars[Var] = Val
▷ Both functions are called at the end of a loop body
procedure INFERINDUCTIONVARS(L)
    for V in L.UpdatedVars.Keys do
        if L.Iteration == 1 then
            L.IVCandidates[V]=L.UpdatedVars[V]
        if L.Iteration == 2 then
            L.IVDeltas[V]=L.UpdatedVars[V]-
L.IVCandidates[V]
            L.IVCandidates[V]=L.UpdatedVars[V]
        if L.Iteration == 3 then
            if L.UpdatedVars[V] - L.IVCandidates[V]
                == L.IVDeltas[V] then
                IVs.append(V)
    return IVs
procedure INFERLOOPITERATIONS(L)
    for C in L.LoopConditions do
        if L.Iteration == 1 then
            C.Value = C.LHS - C.RHS
        if L.Iteration == 2 then
            C.Delta = (C.LHS - C.RHS) - C.Value
        if L.Iteration == 2 then
            if (C.LHS - C.RHS) - C.Value == C.Delta then
                if L.Operator == < then
                    C.LoopCount = -(C.InitialVal / C.Delta)
                if L.Operator == > then
                    ...
```

Figure 5: Functions added for generating loop invariants automatically. ReadLoop and UpdateLoop track the changed variables inside the loop. ReadLoop returns a fresh symbolic variable if a variable is read before written. InferInductionVars and InferLoopIterations track the delta of the variables and loop conditions to find the induction variables, and compute the number of iterations of a loop.

Finding Induction variables. ObliCheck figures out the difference of each variable between the first and second iterations, and the second and third ones. Then ObliCheck checks that the two differences are the same. The first iteration starts with an empty state mapping. When a variable is modified in the first iteration, an entry from the variable to its concrete or symbolic value is updated. If a variable is referenced but does not have an entry in the mapping, an unconstrained symbolic variable is assigned to the referenced variable. This over-approximation takes any possible modifications in previous iterations into account. At the end of the first iteration, the values of the updated variables are saved. The second iteration is executed with the state created during the first iteration. At the end of the second iteration, the difference of the values saved at the first iteration and the second one is calculated and saved. After the third iteration, another set of the deltas is obtained and the variables whose deltas are the same are judged as induction variables.

Calculating the number of iterations. The number of loop iterations depends on the loop condition that bounds the loop. Loop conditions are the conditional statements inside a loop that have one of their targets point to the outside of the loop. A conditional predicate of the form $LHS \circ RHS$ in a loop condition, where $\circ$ is one of the conditional operators ($<, \leq, >, \geq, =, \neq$), can be transformed to $LHS - RHS \circ 0$ and the delta of $LHS - RHS$ between iterations are obtained in the same way that the delta of induction variables are figured out [29]. When the operator $\circ$ is $<$, the number of iterations is $-(InitialValue/Delta)$. Since there can be multiple loop conditions if a loop body has `break` or `return` statements, ObliCheck computes the number of iterations for each loop condition and takes the minimum among them.

After getting the delta per iteration of induction variables and the number of iterations, the loop's post-condition becomes $\bigwedge_i^n IV_i = C_i + D_i * IC_l$, where IV_i represents the induction variables, C_i is each induction variable's initial value before the loop, and IC_l is the number of iterations of the loop l. For example, the algorithm in Listing 4 has two induction variables, `i` and `buf.length`. The post-condition becomes $i = 0 + 1 * inputSize \wedge buf.length = 0 + 1 * inputSize$. The pre-condition of the loop is the loop condition $i < inputSize$, so the loop is summarized as $(i < inputSize) \wedge (i = inputSize \wedge buf.length = inputSize)$.

Limitation. ObliCheck cannot summarize the side-effects of a loop on non-induction variables (*e.g.*, `sum += x`, where `x` is a symbolic expression). Also, if the loop condition depends on a non-induction variable, ObliCheck is unable to infer the number of loop iterations (*e.g.*, `for (i=0; i<y; i+=x)`, where `x` is a symbolic expression, not a constant). The same limitation applies to the recursive functions bounded by input-dependent variables. In these cases, ObliCheck simply assigns an arbitrary symbolic variable to non-induction variables and variables changed in a loop bounded by non-induction variables for over-approximation. If a part of the over-approximated variables is included in the verification condition, it will result in a false-positive. However, in §7 we show that this is not the case for existing oblivious algorithms since the relevant variables such as the length of the output buffer increment by a fixed amount per iteration.

7 Evaluation

7.1 Implementation

We implemented ObliCheck using Jalangi [61], a dynamic program analysis framework for JavaScript. We chose Javascript as a modeling language mainly to leverage the existing open-source Jalangi framework and MultiSE implementation. Other open-source tools such as KLEE [16] and Manticore [52] do not support full-fledged state merging for general programs. Moreover, the idea of value summary representation and incremental state merging is most straightforward to base the implementation of our techniques on. The main concern of our evaluation is the relative performance improvement from our techniques. Hence, we did not consider the absolute performance of existing tools when choosing MultiSE as a baseline.

Overcoming limitations of symbolic execution. We address two challenges posed by the limitation of symbolic execution. First, handling memory address and pointer values can be prohibitively expensive. When references and pointers with symbolic values are de-referenced, symbolic execution invokes a constraint solver to figure out all possible pointer values under the path condition. Finding all satisfying assignments using a constraint solver is prohibitively expensive. We eluded this issue since ObliCheck is based on MultiSE. MultiSE does not require constraint solving for de-referencing pointers because it maintains the set of possible memory addresses of a pointer in the value-summary. This allows ObliCheck to read and write memory locations directly instead of using a constraint solver to reason about memory operations. Second, symbolic execution cannot precisely handle programs with unbounded loops or recursions. Existing tools sacrifice soundness and limit the depth of path to handle this issue. We implemented our loop summarization technique in § 6 to preserve soundness and avoid false-negative cases. ObliCheck is still not able to summarize all unbounded loops as we pointed out in the last paragraph of § 6.

7.2 Evaluation Setup and Input Algorithms

We measured the total analysis time including the symbolic execution and constraint solving time, but excluded the instrumentation time which is syntax-based and done before the symbolic execution. The experiment was done on an AWS instance with Ubuntu 18.04.2, with 2.5 GHz Intel Xeon Platinum 8175 processors and the memory size is 32GB

We evaluate ObliCheck using existing data processing algorithms from data processing frameworks used in production and published academic papers. Table 6 lists the benchmark algorithms. Opaque [78] is an open-source, distributed data analytics frameworks based on Apache Spark [2]. Signal Messenger [7] is an open-source encrypted messaging service commercialized by Signal Messenger LLC. The input programs are derived from either the implementation or written description of the algorithms. However, ObliCheck does not verify the actual implementation of the algorithms and the

Algorithm	Description
Tag	The algorithm in Listing 1
Tag (Not Oblivious)	The algorithm in Listing 1 with the false branch in the if statement removed
Tag&Apply	The algorithm in Listing 3
Sort	Oblivious operator from Opaque project [3]
Filter	Oblivious operator from Opaque project [3]
Aggregate	Oblivious operator from Opaque project [3]
Join	Oblivious operator from Opaque project [3]
MapReduce	MapReduce algorithm by Ohrimenko *et al*. [57]
Decision Tree	Oblivious decision tree inference by Ohrimenko *et al*. [58]
Hash Table	Oblivious hash table used in the Signal messenger contact discovery service [6]
AES Encryption	AES CBC encryption from AES-JS [1]
Neural Net Inference	Prediction part of a neural network from neuroJS [5]
TextSecure Server	End-to-End message encryption server in Javascript [4]

Table 6: List of benchmark algorithms. Tag and Tag&Apply are the example algorithms showed earlier. Sort, Filter, Aggreate and Join are from the Opaque framework [3], MapReduce and Decision Tree are from Ohrimenko *et al*. [57,58] and Hash Table is from the Signal Messenger [7].

input programs are all re-written in the subset of Javascript using ObliCheck APIs.

7.3 Accuracy Test

Example	Oblivious?	Taint Analysis	ObliCheck	
			OSM	OSM+ISU
Tag	○	× ✗	○ ✓	○ ✓
Tag (NO)	×	× ✓	× ✓	× ✓
Tag&Apply	○	× ✗	× ✗	○ ✓
Sort	○	× ✗	○ ✓	○ ✓
Filter	○	× ✗	○ ✓	○ ✓
Aggregate	○	× ✗	○ ✓	○ ✓
Join	○	× ✗	○ ✓	○ ✓
MapReduce	×	× ✓	× ✓	× ✓
DecisionTree	○	× ✗	○ ✓	○ ✓
HashTable	○	× ✗	○ ✓	○ ✓
AES Encryption	○	○ ✓	○ ✓	○ ✓
Neural Net Inference	○	○ ✓	○ ✓	○ ✓
TextSecure Server	×	× ✓	× ✓	× ✓

Table 7: Accuracy evaluation result of each technique over the benchmark suite algorithms. Taint Analysis checks the algorithm has a secret dependent branch by taint tracking. OSM is our optimistic state merging technique where only the `length` of buffers are not merged, and ISU is our iterative state unmerging technique (ObliCheck). ○ means the algorithm is classified as oblivious and × represents one is classified as not oblivious. ✓ marks the test result is correct (either true positive or true negative)and ✗ marks the result is an error (either false positive or false negative).

We first evaluate the accuracy of ObliCheck's techniques (i.e., optimistic state merging and iterative state unmerging) and compare it with other existing techniques – namely, taint tracking, and symbolic execution with conventional state merging (MultiSE). Table 7 displays the results. MapReduce is not oblivious because it pads the output up to the possible maximum length of the output based on the input data. Thus, it leaks information regarding the input data distribution. TextSecure Server is not oblivious since the server sends the different lengths of the messages based on the status of the devices and it does not pad the messages before sending them.

Taint analysis classifies all algorithms as not oblivious except for AES Encryption and Neural Net Inference. Both of the two are only algorithms without secret-dependent branches. Our optimistic state merging technique obtains the correct results except for the Tag&Apply (in Listing 3) example. As we discussed in §5, optimistic state merging enables two paths with different symbolic states to get merged precisely by the overapproximation. However, the performance improvement comes at the cost of accuracy due to false positives. In Tag&Apply, simply merging all the tag values leads to false positive because of the `if` statements in the `apply` function in Listing 3.

With iterative state unmerging, ObliCheck iterates the symbolic execution in addition to the first Optimistic State Merging phase. In Listing 3 with the Tag&Apply source code, the tag value is unmerged in the second iteration then ObliCheck correctly classifies the program as oblivious. Both conventional state merging and our iterative state unmerging technique correctly identify oblivious and non-oblivious algorithms. There is no false-negative case in either technique. We discuss the cost of additional iterations of iterative state unmerging in the next evaluation.

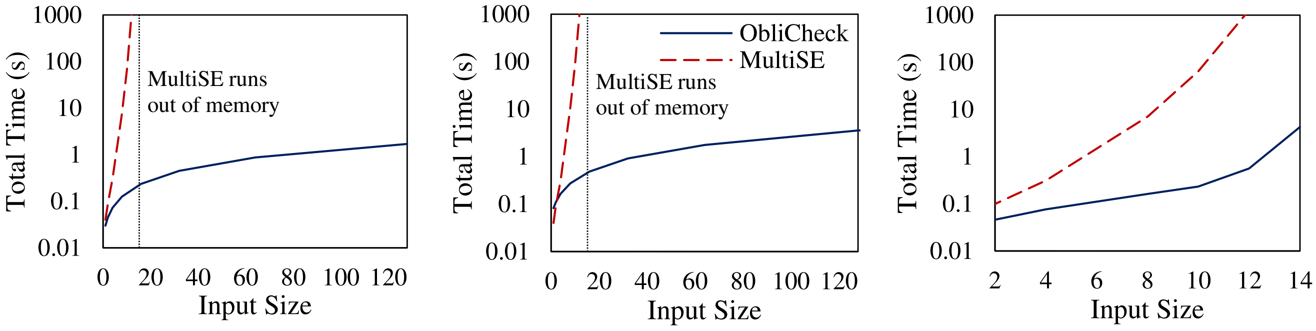
7.4 Performance Evaluation

Pure symbolic execution suffers from path explosion and conventional state merging does not fully address this issue. We evaluate the performance of applying conventional state merging to ObliCheck and show how much performance improvement it achieves in terms of total program analysis time. We also measured the overhead of iterative state merging compared with a non-iterative domain-specific merging technique. We set the length of the input data as large as possible until MultiSE is on the brink of out of memory. The input data to be processed is considered private in all the examples. In Neural Net Inference, we consider the size of the network layers is not private. In TextSecure Server, we consider the destination device addresses are private input.

Table 8 shows the evaluation results of pure MultiSE and ObliCheck on the test algorithms. ObliCheck performs up to 50300× faster than MultiSE. The improvement mainly comes from the reduced number of exploration paths and simplified path conditions due to optimistic state merging.

Example	LoC	Branch	Symbolic Execution (MultiSE)		ObliCheck (OSM)		ObliCheck (OSM + ISU)			
			Total Time (s)	Avg Value Summary Size	Total Time (s)	Avg Value Summary Size	Total Time (s)	Avg Value Summary Size	Speed Up (×) (vs MultiSE)	Overhead (%) (vs OSM)
Tag	27	90	5176.90	459.52	0.19	1.37	0.20	1.37	26548.16	0.52
Tag (NO)	25	90	5141.39	589.71	1.48	2.23	1.49	2.23	3450.60	0.02
Tag&Apply	32	94	5148.17	377.00	0.27	1.43	0.46	1.42	11167.40	70.74
Sort	149	263	4614.00	4.01	0.44	1.60	0.45	1.60	10276.16	0.02
Filter	150	287	14970.46	7.46	0.41	1.58	0.42	1.58	35900.39	0.01
Aggregate	156	268	4875.15	3.99	0.35	1.61	0.34	1.61	14380.99	-0.01
Join	160	268	3912.44	4.06	0.31	1.61	0.31	1.61	12620.77	-0.02
MapReduce	62	241	8154.90	204.86	9.68	2.09	38.32	2.20	212.79	296.11
DecisionTree	35	653	9305.51	465.12	0.19	1.01	0.19	1.01	50300.04	1.64
HashTable	42	139	1683.32	38.64	0.15	1.39	0.16	1.39	10520.75	0.0
AES Encryption	754	0	1.00	1	0.99	1	1.00	1	1.00	0.0
Neural Net Inference	179	0	4.84	1	4.84	1	4.78	1	1.03	-0.01
TextSecure Server	158	149	3433.11	36.23	0.18	1.44	0.18	1.44	19506.32	0.02

Table 8: Performance evaluation result of each technique on the test algorithms. OSM refers to optimistic state merging, and ISU to iterative state unmerging. LoC is the lines of code of each program. Branch refers to the number of branches encountered during a single execution. The total time includes the execution time of the symbolic execution engine and the solver time of ObliCheck. The average value summary size is the average length of the value summary, which reflects how efficiently state merging was done. OSM shows the best performance since it merges everything and executes a program only once. ObliCheck with ISU has less than 1.64% of the overhead for the test algorithms except for Tag&Apply and MapReduce. Two algorithms are a false positive and a true negative, which make ObliCheck iterates more.



(a) Oblivious Tagging (True Negative) **(b)** Oblivious Tag and Apply (False Positive) **(c)** Non-oblivious Tagging (True Positive)

Figure 6: Total analysis time of MultiSE (conventional state merging) and ObliCheck (domain-specific merging followed by iterative state unmerging) over Tag, Tag&Apply, and Tag (Non-oblivious). The total time of MultiSE grows exponentially until the input size 16 and fails to finish due to out of memory error after then when it analyzes Tag and Tag&Apply. The total analysis time of ObliCheck grows linearly without out of memory error. The total time of ObliCheck blows up exponentially when it checks the non-oblivious Tag algorithm. This is because state merging is not possible after unmerging merged state and the size of state exponentially grows as MultiSE does.

The overhead of iterative state merging is marginal if the algorithm is oblivious. If the algorithm is not oblivious (true positive) or needs more iterations to turn out to be oblivious (false positive) it becomes more significant. In Tag&Apply, a false-positive case, the overhead is 70.74%. The additional iteration of iterative state unmerging causes this extra execution to report a correct result. The maximum overhead is 296% for checking MapReduce in the benchmark suite. Although iterative state unmerging costs some performance improvement achieved by optimistic state merging for true-positive cases, ObliCheck achieves a significant improvement over conventional symbolic execution. In MapReduce, ObliCheck still achieves 212.79× of speedup. This is because ObliCheck only unmerges the variables affecting the verification condition instead of re-running a program without any merging.

We also demonstrate the scalability of ObliCheck compared with conventional state merging techniques, by running vanilla MultiSE and ObliCheck over Tag, Tag&Apply and Non-oblivious Tag algorithms. The algorithms result in a true negative, false positive and true positive respectively when checked using optimistic state merging.

Figure 6 shows the results. ObliCheck boasts linear scalability when it checks Tag, and Tag&Apply algorithms, which are oblivious. In contrast, the runtime of MultiSE grows exponentially for non-oblivious Tag since it fails to merge the states in the end. In this case, ObliCheck provides the information regarding the program statements where state unmerging has been applied so that an algorithm designer can manually inspect and judge a given algorithm is truly non-oblivious.

Table 9 demonstrates the loop summarization performance of ObliCheck. The number of loops only include ones summarized by ObliCheck. For example, AES Encryption algorithm contains multiple for loops but only one outermost loop has the input length in its loop condition. All the other loops are

Example	MultiSE	ObliCheck	# of Loops	Total Time (s)
Tag	∞	◯ ✓	2	0.060
Tag (NO)	∞	× ✓	2	0.062
Tag&Apply	∞	◯ ✓	2	0.138
Sort	∞	◯ ✓	30	0.245
Filter	∞	◯ ✓	34	0.290
Aggregate	∞	◯ ✓	30	0.161
Join	∞	◯ ✓	30	0.160
MapReduce	∞	× ✓	26	0.439
DecisionTree	∞	◯ ✓	5	0.117
HashTable	∞	◯ ✓	6	0.151
AES Encryption	∞	◯ ✓	5	0.017
Neural Net Inference	∞	◯ ✓	5	0.016
TextSecure Server	∞	× ✓	2	0.065

Table 9: Loop invariant generation test result. The # of Loops column includes the number of loops summarized by ObliCheck. ∞ means the checking process runs infinitely. MultiSE runs infinitely for all test algorithms because of input-dependent loops. ObliCheck classifies each algorithm correctly by summarizing the loops.

constants. As we discussed in §4.1, MultiSE runs infinitely when a given algorithm contains input-dependent loops and thus cannot verify it. In contrast, ObliCheck generates loop invariants automatically and classifies every test algorithm correctly within a second.

7.5 Case Study on the Applications

ObliCheck boasts the biggest speedup on the Decision Tree application. The code in Listing 5 is from the application. A decision tree compares a given input and intermediate decision nodes to provide a prediction result. The oblivious decision tree keeps accessing the rest of the layers even after finding a leaf node to keep the visible access patterns the same regardless of the input value. Regular symbolic execution suffers from the path explosion since it diverges at every iteration due to the branch statement. In contrast, ObliCheck merges the branch statement and correctly judges the obliviousness of a program with the orders of magnitude speedup.

```
var cur = 0, found = 0;
for (var i = 0; i < layerLen; i++) {
    if (privateData < layers[i][cur]) {
        cur = cur * 2;
    } else if (privateData > layers[i][cur]) {
        cur = cur * 2 + 1;
    } else {
        found = cur;
        cur = cur * 2;
    }
}
```

Listing 5: A branch statement from Decision Tree.

ObliCheck accomplishes a speedup on the Hash Table application similarly. In Listing 6 from Hash Table, ObliCheck merges the if statement that calculates the index of a bucket. The x variable is used to calculate the remainder based on the length of the cache line. This modulo operation figures out at which index privateData[i] should be inserted. This merging prevents the path explosion problem.

```
for (var i = 0; i < cacheLineLen; i++) {
    for (var j = 0; j < dataLen; j++) {
        var x = readSecretInput();
        if (x * cacheLineLen + i ==
             privateData[j]) {
            cacheLines[i][nextAvailableCache[i]] =
                privateData[i];
            nextAvailableCache[i] += 1;
        } else {
            cacheLines[i][dummySlot] =
                privateData[i];
        }
    }
}
```

Listing 6: A branch statement from Hash Table

8 Discussion

8.1 Generalization for Checking Other Side Channels

ObliCheck proves the absence of the access pattern side-channel by keeping the access sequence as a program state. Based on the recorded state, ObliCheck checks whether the predefined verification condition holds at the end of symbolic execution. The oblivious property enforced by ObliCheck guarantees the absence of the access pattern based side-channel leakage at the algorithm level. In principle, other types of side-channel leakage can also be verified similarly. For example, one can model timing side-channels by recording the number of steps of an algorithm while symbolically executing an algorithm. In contrast to existing works that rule out algorithms with secret dependent branches and memory accesses entirely [15, 69], comparing the time it takes to finish each execution path directly is a more precise approach. By (1) modeling observable behavior of an algorithm as program state during the symbolic execution, and (2) defining the verification condition based on the state, one can prove the side-channel leakage using the same technique used in ObliCheck. We leave the generalization of our technique for different types of side-channels as future work.

8.2 Checking Probabilistically Defined Obliviousness

ObliCheck checks if a given algorithm has the same deterministic access sequence across all possible inputs. In contrast, the original ORAM work defines obliviousness probabilistically. To verify the obliviousness condition in this case, a checker should keep the probability distribution of access sequences and verify the distributions of any two inputs are indistinguishable. For this, a symbolic execution engine should be able to capture how a variable with probability distribution is transformed over the algorithm execution. Several techniques have been proposed recently to automatically verify differential privacy, which certifies the distance between any two algorithm outputs is within a concrete bound [9, 13, 76]. For example, LightDP [76] provides a language with a lightweight dependent type incorporating probability distribution. Similarly, ObliCheck can be extended with APIs or with a new domain-specific language (DSL) to capture probability distribution, and its transformation during the execution. The final verification condition checks the statistical distance of the observable state for any two inputs. This interesting direction requires further investigation and we leave it for future work.

8.3 Checking Algorithms in a Different Programming Language

Although core techniques of ObliCheck can be implemented in any other languages, we found Javascript is the right choice as a modeling language in most cases. It is a dynamic language that does not require static typing, compilation, and explicit memory management. These characteristics facilitate rapid prototyping of an algorithm to quickly check its obliviousness using ObliCheck. Although it has some quirks such as the unusual semantics of equality, ObliCheck utilizes a clean subset of Javascript as modeling language and thus clear enough for modeling algorithms. However, we found it unnatural to model the low-level behavior of an algorithm in Javascript. A user has to write assembly-like code using Javascript to verify the machine code-level obliviousness. Instead of using Javascript, it is natural to devise a Domain Specific Language for writing an algorithm in this case. Then a compiler translates it to an intermediate representation such as LLVM IR for verification, rather than a user manually describes the low-level behavior of an algorithm.

9 Related Work

Checking Side Channel Leakage Using Taint Analysis. Several past works detect or mitigate side-channel leakage of an algorithm using taint analysis. Vale [15] provides a DSL and tools for writing high-performance assembly code for cryptographic primitives. Vale checks the written code is free from digital side-channels of memory and timing using taint analysis. As described in §2.4, this approach can result in a large number of false positives in the presence of unobservable state.

Raccoon [59] uses taint analysis to identify secret dependent branches which can potentially leak information and obfuscate the behaviors of these branches. Since Raccoon is a compiler but not a checker, using taint analysis in this way may result in unnecessary obfuscation but not the rejection of a program. Sidebuster [77] uses taint analysis in the same way to check and mitigate side-channels in web applications. Overall, taint analysis is an efficient technique to detect and mitigate side-channels under a limited time budget. However, it keeps a coarse-grained state regarding information flow and only tracks which variables are affected by a source input.

Symbolic Execution and State Merging Techniques for Preventing Side Channel Attacks. Symbolic execution has widely deployed to check certain properties of a program and generate high-coverage test cases [16–18, 28, 38, 61, 62]. Practical symbolic execution frameworks normally limit the depth of exploration or drive the execution to parts of a code to find buggy code with a limited time budget. Our checker rather checks the whole input space of a program to eliminate false-negative cases to make our checker useful for checking the security property.

State merging techniques are used to resolve the path explosion problem of symbolic execution at the expense of more complicated path conditions [10, 27, 30]. MultiSE [63] merge states incrementally at every assignment of symbolic variables without introducing auxiliary variables. MultiSE supports merging values not supported by constraint solver such as functions and makes it unnecessary to identify the join points of branches to merge state. OSM of ObliCheck is fundamentally different from existing state merging techniques. Existing state merging techniques are sound and complete with regard to symbolic execution. The merged symbolic state explores the same set of program behaviors as regular symbolic execution. Therefore, existing techniques do not report false positives. In contrast, OSM leverages domain-specific knowledge from oblivious programs and over-approximates program behavior to merge two states even if they cannot be merged in original state merging, which significantly speeds up the checking process. However, OSM might report false positives, and that's where ISU kicks in to repair them.

One of the most widely exploited and studied side-channels is the cache side-channel. CaSym [45] uses symbolic execution to detect a part of a given program that incurs cache side-channel leakage. CaSym runs the LLVM IR of a program symbolically and finds inputs which let an attacker distinguish observable cache state. CaSym merges paths by introducing an auxiliary logical variable. CaSym and ObliCheck differs in how they merge state—CaSym merges at join points by introducing auxiliary variables, while ObliCheck merges at each assignment statement. Moreover, CaSym does not use domain-specific knowledge to merge state aggressively. CaSym specifically focuses on checking cache side-channel leakage with a comprehensive cache model but ObliCheck is for more general oblivious algorithms. CacheD [69] also uses symbolic execution but only checks the traces explored in a dynamic execution of a program, which may miss potential vulnerabilities. CacheAudit [21] uses abstract interpretation to detect cache side-channel leakage.

Ensuring Noninterference Policy. Noninterference is a security policy model which strictly enforces information with a 'high' label does not interfere with information with a 'low' label [20]. Some existing approaches for enforcing noninterference are type checking [53, 54, 68] and abstract interpretation [25, 39]. Barthe *et al.* defined a way to prove noninterference by a sequential composition of a given algorithm [12]. Terauchi and Aiken proposed a term 2-safety to distinguish safety property like noninterference which requires observing two finite sets of traces [67]. Also, they devised a type-based transformation of a given algorithm for self-composition which has better efficiency than a simple sequential-composition suggested by Barthe *et al.* for removing redundant and duplicated execution. Milushev *et* *al*. suggested a way to use symbolic execution to prove the noninterference property of a given algorithm [50]. They used type-directed transformation suggested by Terauchi and Aiken to interleave two sets of algorithms. The type-directed transformation can be orthogonally applied and potentially improve the performance of ObliCheck.

10 Conclusion

Access pattern based side-channels have gained attraction due to a large amount of information they leak. Although oblivious algorithms have been devised to close these side-channels, the algorithms must be manually checked. We showed that symbolic execution can be utilized to automatically check a given algorithm is oblivious. With our optimistic state merging and iterative state unmerging techniques, ObliCheck achieves more accurate results than existing taint analysis based techniques and runs faster than traditional symbolic execution.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Vasileios P. Kemerlis, for their feedback. This work was supported in part by the NSF CISE Expeditions CCF-1730628, NSF Career 1943347, NSF grants CCF-1900968, CCF-1908870, CNS-1817122, and gifts/awards from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Fujitsu, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Aes-js: A pure javascript implementation of the aes block cipher algorithm and all common modes of operation. `https://github.com/ricmoo/aes-js`. Accessed: 2020-08-12.

[2] Apache spark: Lightning-fast unified analytics engine. `https://spark.apache.org/`. Accessed: 2020-08-10.

[3] Github: Opaue. `https://github.com/ucbrise/opaque`. Accessed: 2020-02-10.

[4] A javascript implementation of a textsecure server. `https://github.com/joebandenburg/textsecure-server-node`. Accessed: 2020-08-12.

[5] neurojs: Neural network library. `https://github.com/pieteradejong/neuroJS`. Accessed: 2020-08-12.

[6] Private contact discovery service (beta). `https://github.com/signalapp/contactdiscoveryservice`. Accessed: 2020-08-31.

[7] Technology preview: Private contact discovery for signal. `https://signal.org/blog/private-contact-discovery/`. Accessed: 2019-05-06.

[8] Mohsen Ahmadvand, Anahit Hayrapetyan, Sebastian Banescu, and Alexander Pretschner. Practical integrity protection with oblivious hashing. In *ACSAC*, 2018.

[9] Aws Albarghouthi and Justin Hsu. Synthesizing coupling proofs of differential privacy. In *POPL*, 2017.

[10] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS/ETAPS*, 2008.

[11] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD*, 2013.

[12] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSF*, 2004.

[13] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. In *CCS*, 2016.

[14] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[15] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security*, 2017.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[17] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *CACM*, 2013.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[19] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. In *PET*, 2017.

[20] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *CACM*, 1977.

[21] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *USENIX Security*, 2013.

[22] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, 2010.

[23] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[24] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[25] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *POPL*, 2004.

[26] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[27] Patrice Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[29] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, 2011.

[30] Patrice Godefroid, Aditya Nori, Sriram Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.

[31] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[32] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[33] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Jean-Louis Lanet, and Routa Moussaileb. Detection of side channel attacks based on data tainting in android systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2017.

[34] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[35] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, 2006.

[36] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *CoRR*, abs/1803.05961, 2018.

[37] S. Kadloor, X. Gong, N. Kiyavash, T. Tezcan, and N. Borisov. Low-cost side channel remote traffic analysis attack in packet networks. In *IEEE International Conference on Communications*, 2010.

[38] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[39] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *CCS*, 2013.

[40] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP*, 2007.

[41] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *PLDI*, 2012.

[42] Dayeol Lee, Dongha Jung, Ian Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security*, 2020.

[43] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[44] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[45] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang. Deepsec: A uniform platform for security analysis of deep learning model. In *IEEE S&P*, 2019.

[46] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[47] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEE S&P*, 2015.

[48] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *CCS*, 2017.

[49] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[50] Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. In *FMOODS/FORTE*, 2012.

[51] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *IEEE S&P*, 2018.

[52] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE*, 2019.

[53] Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, 1999.

[54] Andrew C. Myers, Andrew C. Myers, and Barbara Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[55] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE S&P*, 2011.

[56] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *CCS*, 2013.

[57] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *CCS*, 2015.

[58] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[59] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[60] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *IACR Cryptology ePrint Archive*, 2017:549, 2017.

[61] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE 2013*, 2013.

[62] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *ESEC/FSE-13*, 2005.

[63] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *ESEC/FSE 2015*, 2015.

[64] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In *CCS*, 2017.

[65] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *International Conference on Fundamental Approaches to Software Engineering*, 2008.

[66] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS*, 2013.

[67] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[68] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[69] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *USENIX Security*, 2017.

[70] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, 2014.

[71] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[72] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, 2016.

[73] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.

[74] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP*, 2009.

[75] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[76] Danfeng Zhang and Daniel Kifer. Lightdp: Towards automating differential privacy proofs. In *POPL*, 2017.

[77] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *CCS*, 2010.

[78] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.