obliv-clang: Real-World Oblivious Programming in C++

Yunqian Luo[1][0009−0000−3366−3722] and Mingyu Gao[1,2][0000−0001−8433−7281]✉

[1] Tsinghua University, Beijing, China
[2] Shanghai Qi Zhi Institute, Shanghai, China
luoyq23@mails.tsinghua.edu.cn, gaomy@tsinghua.edu.cn

Abstract. Side-channel vulnerabilities, particularly timing and access-pattern-based attacks, have become critical issues for confidential data processing in trusted environments. Oblivious programming is an effective approach to alleviate these attacks by making program execution not leak any secret through execution time and data access traces. To facilitate oblivious programming in practice, we propose a compilation-time checking tool, obliv-clang, which can comprehensively check the obliviousness of a program written in C++. It is designed to support the rich language features in C++, including the complicated concept of arbitrarily nested pointers, in order to seamlessly work with existing industry-level codebases and produce high-performance compiled binaries with minimum compilation overheads. We design a set of rules in obliv-clang and formally prove their soundness in the presence of complicated C++ language features. We also implement several non-trivial oblivious algorithms as case studies to demonstrate the expressiveness of obliv-clang, and show that programs compiled using obliv-clang can outperform previous solutions.

Keywords: Compiler · Trusted Execution Environment · Oblivious Programming

1 Introduction

Confidential data processing in trusted environments, such as trusted execution environments (TEEs), remains vulnerable to various side-channel attacks exploiting shared caches [10], OS-controlled page tables [4, 22], branch predictors [12], and speculative execution [31]. These attacks succeed because certain sensitive execution properties leak through timing observation and shared hardware units, even when data are architecturally isolated. The major defenses to these side-channel attacks can be categorized into hardware changes and software techniques. They complement each other—timing and I/O pattern issues are best avoided by software, while speculation-based vulnerabilities require hardware fixes. This work focuses on the software solutions, among which the most prominent approach is *oblivious programming*: making execution traces (timing, I/O patterns) independent of secrets.

Existing methods for oblivious programming each have drawbacks. Domain-specific languages (DSLs) [5,15,33] require costly toolchain integration, especially in industrial environments with complicated software projects. Separate modeling tools [25, 34] face consistency issues between models and implementations. Compiler transformations [19, 24] lack high-level application semantics and can only be done conservatively, resulting in performance degradation.

In this work[3], we present obliv-clang, a compiler plugin for the widely used, industry-class C++ language, which checks the obliviousness of a program statically at the compilation time. It is implemented using the popular `clang` and LLVM infrastructure. As the key principle, obliv-clang is designed to work directly with C++ and support its rich language features, including pointers and references, function calls, compound types, object-oriented programming, templates, and more. Therefore, it can seamlessly work with existing codebases with minimum porting effort, eliminating the extra burden of using a new domain-specific language in previous work. In addition, obliv-clang is implemented directly in the compiler, so it avoids the consistency issue between separate modeling and implementation. Finally, using a familiar, mature, and industry-class language like C++ allows developers to conveniently write their code, and also able to retain high performance for the developed oblivious programs.

The key challenge of designing a compilation-time checker like obliv-clang is its soundness, meaning that the check must be comprehensive so that if a program passes, it should be considered safe to execute. The major difficulty is to handle the arbitrarily nested pointers (like "a pointer to a pointer to a pointer") in C++, which can be dereferenced, taken-address, and assigned in many ways. In particular, we find that assigning a secret pointer with a public pointer value, which seems benign, could sometimes be unsafe and leak sensitive data (see Example 2 in Section 3.2). To address this problem, obliv-clang formally introduces the concept of oblivious levels (`olevel`s), together with a set of rules to deduce each variable's `olevel` along the program dataflow and to forbid unsafe operations accordingly. These rules are extended beyond simple operations to support also function calls, compound types, and templates. In such a way, obliv-clang is able to provide verifiable security with formal proofs.

Since obliv-clang is a front-end checker, we also need to ensure that the backend compiler optimization passes do not break the obliviousness property. We thoroughly analyze the relevant optimization passes in LLVM and suppress the ones that may compromise obliviousness. Furthermore, we apply an extra double-check on the generated assembly code using existing tools [8], which ensure end-to-end obliviousness.

We have implemented several non-trivial oblivious algorithms, including PathORAM [28] and Oblivious BFS [3], using our obliv-clang tool, in order to demonstrate its good expressiveness. We also quantitatively evaluate obliv-clang, showing that the extra obliviousness check incurs minor overheads (less than 18%) to the compilation time, and it produces relatively high-performance compiled binaries. When compared to previous tools, the programs compiled with obliv-clang

[3] An extended version of this paper is available at [16].

run 27.8% faster than Jasmin [1] and 66.9% faster than FaCT [5] on average. The implementation of obliv-clang has been open sourced at github.com/tsinghua-ideal/obliv-clang.

Our contributions in this paper are summarized below:

- The design and implementation of a program obliviousness check tool, obliv-clang, as a compiler plugin working seamlessly with the C++ language.
- A set of obliviousness check rules that support the rich set of C++ language features, including arbitrarily nested pointers, function calls, compound types, templates, etc.
- A formal proof about the soundness of our obliviousness check rules.
- The case studies and evaluation experiments to demonstrate the expressiveness and performance advantages of obliv-clang.

2 Background and Motivation

2.1 Oblivious Programming

Program obliviousness prevents attackers from learning sensitive information through execution traces (timing, memory, and I/O patterns). Attackers extract traces either directly (through privileged access) or indirectly via shared hardware [4, 10, 22]. Oblivious programming requires branch conditions, memory, and I/O patterns to remain independent of sensitive data. Common techniques include redundant branch execution, oblivious memory scanning, and `cmove`-like instructions. Researchers have developed efficient algorithms including ORAM [28], oblivious BFS [3], and oblivious data structures [32].

Oblivious programming is tricky and subtle—developers can easily break obliviousness through overlooked details. Thus, automatic tools are essential to facilitate correct oblivious programming.

2.2 Tradeoffs and Limitations of Existing Tools

Previous work can be classified along two dimensions: (1) whether behaviors are modified, i.e., check-only [1, 20, 25] vs. transforming [5, 14, 15, 19, 23, 24]; and (2) at which level to perform analysis, e.g., the frontend abstract syntax tree (AST) [5, 14, 24, 25] vs. the backend intermediate representation (IR) [19, 23]. Automatic transformations could introduce performance overheads due to conservative handling. Frontend approaches offer better analysis but backend optimizers may undermine their security. Additionally, many designs rely on DSLs, which require costly toolchain development and lack standard features.

2.3 Our Approach

We build upon C++ and the `clang` compiler, verifying obliviousness directly on source code at compilation time. C++ is widely used in industry. It offers well-defined semantics in its specification [29], a minimal runtime with close hardware

control, and compatibility with numerous software interfaces and libraries. The `clang` compiler offers a friendly, extensible plugin interface used by many tools [6, 7], enabling non-intrusive checker implementation. The `clang` frontend processes C++ and C code in a unified AST format, allowing obliv-clang to handle C code seamlessly—important since many system and cryptographic libraries are written in C. While our tool is a checker, we also implement backend mechanisms to guarantee obliviousness during code generation (Section 5).

Challenges. Working directly with C++ presents challenges due to its rich semantics. First, expressing secrecy for arbitrarily nested pointers requires a precise scheme. Second, function calls across translation units need obliviousness-aware interfaces. Third, object-oriented features require handling relationships between class instances, members, and methods. Fourth, templates may instantiate with both sensitive and non-sensitive data. Finally, external libraries require annotations invisible to the compiler. Our design in Section 3 addresses these.

2.4 Security Model

Our security model focuses on timing and access-pattern-related attacks in two scenarios:

1. The victim program executes in a TEE enclave, where adversaries with hardware access can perform access-pattern-based attacks, including timing, cache [10], and page-table [4, 22] attacks.
2. The victim program executes on normal hardware, where adversaries without hardware access can interact with the program and observe timing.

Side-channel attacks based on power [13], electromagnetism [21], speculative execution [31], rowhammer [11], and similar vectors are beyond our scope.

To ensure security under these assumptions, programs must satisfy the following properties in Theorems 1 and 2. Formal definitions of object secrecy will be provided in Section 3.

Theorem 1 (Secrecy). *For a program P with no undefined behaviors, and any external input x, let $v_{pub}(x)$ be the set of values of its public objects during execution, $\mathsf{Trace}(P, x)$ be the execution trace. There exists a probabilistic polynomial time (p.p.t.) simulator Simu such that no p.p.t. adversary $\mathcal{A}$ can distinguish $\mathsf{Trace}(P, x)$ and $\mathsf{Simu}(v_{pub}(x))$. Formally:*

$$\forall x, \left| \Pr[\,\mathcal{A}(v_{pub}(x), \mathsf{Trace}(P, x))\,] - \Pr[\,\mathcal{A}(v_{pub}(x), \mathsf{Simu}(v_{pub}))\,] \right| < \epsilon(n), \quad (1)$$

where n is a security parameter, and ϵ is a negligible function.

Theorem 2 (Dataflow (informal)). *There is no data flow from secret objects to public objects.*

3 Design

We designed obliv-clang, a C++ compiler plugin for `clang` that verifies program obliviousness directly on source code at compilation time. As a checker,

obliv-clang's primary goal is *soundness*: programs passing its checks should be considered safe. This section details obliv-clang's semantics, including the required programmer annotations to indicate protected data and the comprehensive rules enforced to ensure obliviousness. We focus on addressing challenges of supporting flexible C++ semantics—pointers, references, function calls, and compound types—while maintaining compatibility with external libraries. The formal soundness proof is provided in the extended version of this paper [16].

3.1 Getting Started: Oblivious Variables

The basic information required by obliv-clang is the knowledge of which variables should be treated as secret. Since a compiler cannot inherently determine which objects are sensitive, we require programmer annotations in the source code to provide this information. We believe such annotations only add moderate burden to programmers, as programmers typically well understand the purposes and secrecy requirements of their variables.

Beginning with a minimal language subset (no references, no nested pointers, only plain values), an object's secrecy can be described by a boolean flag. obliv-clang requires secret variables to be annotated with the custom `obliv` attribute. C++ specifications allow compilers to define custom attributes as language extensions [29, Chapter 9.12], which can be applied to declarations, statements, translation units, and other elements. Furthermore, with the `clang` toolchain, parsing attributes can be easily done using plugins, without intrusive changes to the compiler's internal core logic (Section 4).

To satisfy the Secrecy and Dataflow properties from Section 2.4, the following rules are required:

Rule 1 (Basic Arithmetic) Any expression depending on an oblivious object is also oblivious. Note that we only consider value dependencies; comma expressions, compilation-time constant expressions (e.g., `sizeof`, `alignof`), and take-address operations are not considered dependencies.

Rule 2 (Basic Casting) It is forbidden to cast an oblivious expression to a non-oblivious expression.

Rule 3 (Basic Dereference) It is forbidden to dereference an oblivious pointer. Similarly, it is forbidden to take an oblivious object as the base address or index in an array subscript expression.

Rule 4 (Allocation) It is forbidden to create an object whose size is oblivious (e.g., `new[]` with oblivious length).

Rule 5 (Condition) The condition in any conditional clause (`if`, `while`, `for`, `switch`, ternary expressions, short-circuit boolean expressions) must not be an oblivious expression.

Note that "cast" in Rule 2 refers to both explicit and implicit casting. In assignments, the right side is first cast into the target type before assignment to the left side, so this can also be considered as an assignment rule.

The following code snippet demonstrates these rules:

Example 1.

```
#define OBLIV [[obliv]]  // to make it look less cumbersome

// declare an oblivious variable
OBLIV int x = 1729;
// error: cannot assign oblivious value to public variable
int y = x + 3;
// ok
OBLIV int z = x * 10;
// error: cannot take oblivious value as control-flow condition
if (x > 0) x = x - 1;
// error: cannot take oblivious value as array subscript
int *arr = new int[16]; int w = arr[x % 16];
```

3.2 Obliviousness Levels

For a pointer, obliviousness has dual meanings: the pointer's own value (which memory address it points to) and its dereferenced value (content at that address). With arbitrarily nested pointers in C++, this becomes increasingly complex.

We propose the concept of *obliviousness level* (`olevel`) to describe multi-level obliviousness. An expression's `olevel` is a binary string where the i-th digit (counting from the right, starting from 0) represents the obliviousness after dereferencing it i times. An expression whose `olevel` ends with 1 is considered oblivious. In our discussion, an expression is less/more oblivious if its last `olevel` digit is less/more than another's; an expression has no greater/smaller `olevel` if every digit of its `olevel` is no greater/smaller than the corresponding digit of another; "greater/smaller" is defined as the contrary of "no greater/smaller."

Dataflow occurs with assignment expressions. To satisfy the dataflow property, we should not allow assigning an expression with another of a greater `olevel`. However, assignments with smaller `olevel` can also be dangerous. In Example 2, `ppc` of `olevel 100` is assigned with `&public_c` of smaller `olevel 0`:

While assignments between different `olevel`s appear unsafe, some cases with smaller `olevel`s seem benign. For example, assigning a public value to a secret value, is clearly safe — we can not pass any secret information into public by this assignment. But when pointer nests get deeper, it is much less oblivious whether an assignment is safe.

Example 2.

```
char public_c = 'p';
[[obliv("1")]] char secret_c = 'x';
[[obliv("10")]] char *pc = &secret_c;
[[obliv("100")]] char **ppc = &pc;

// if obliv("00") to obliv("10") is allowed
*ppc = &public_c;
*pc = secret_c;  // 'x' flows to public_c!
```

To formalize the dataflow property, we distinguish between the `olevel` in an expression's type and the *inherent* `olevel` of the underlying memory location. In C++, lvalues correspond to memory areas; their inherent `olevel` is defined by the allocation site (variable declarations for stack/static allocations, or `new` expressions for heap allocations). We call lvalues corresponding to allocations *primitive lvalues*. Two lvalues *alias* when they refer to the same memory area. In a well-formed program, every lvalue aliases with a primitive lvalue. We formalize Theorem 2 as:

Theorem 3 (Dataflow of Assignment). *There is no execution of an assignment statement whose assignee (which must be an lvalue) is aliasing a non-oblivious lvalue, but the assigner is an oblivious value.*

Theorem 4 (Dataflow of Value Dependency). *An expression whose value depends on oblivious values is also oblivious.*

We use the following notations: $x\downarrow^k$ denotes dereferencing x by k times; $\texttt{const}(x)$ and $\texttt{obliv}(x)$ denote the constantness and obliviousness of x.

Now we specify the `olevel` rules to satisfy Theorem 3. The danger in Example 2 occurs because an oblivious expression `**ppc` aliases with a non-oblivious expression `public_c`, but `*ppc` is not constant and can be assigned a secret value which flows to the non-oblivious `public_c`. To avoid this, we restrict "non-oblivious to oblivious" assignments by ensuring the ancestors in the pointer hierarchy are constant:

Rule 6 (Casting) Consider two expressions, a and b. b can be implicitly cast to a if and only if, in addition to C++'s cast rules, the following *oblivious cast rules* hold:

1. For a $k > 0$, if $\texttt{const}(a\downarrow^k) = 0$, for any $k' \geq k$,

$$\texttt{const}(a\downarrow^{k'}) = \texttt{const}(b\downarrow^{k'}) \tag{2}$$

$$\texttt{obliv}(a\downarrow^{k'}) = \texttt{obliv}(b\downarrow^{k'}). \tag{3}$$

2. For any $k \geq 0$,

$$\texttt{obliv}(a\downarrow^{k}) \geq \texttt{obliv}(b\downarrow^{k}). \tag{4}$$

For an assignment $a \leftarrow b$, if a is a reference type, the assignment is allowed when b can be implicitly converted to the `olevel` and type of a according to the above casting rules. If a is a value type, b only needs to be converted to the `olevel` and type of constant a, i.e., with $\texttt{const}(a)$ set to 1 but other `olevel` and type unchanged.

Intuitively, condition 2 in Rule 6 prevents assigning oblivious values to non-oblivious targets, e.g., assigning an oblivious b to a non-oblivious a. Condition 1 parallels C++'s implicit conversion rule for `const`-modifiers [29, Chapter 7.3.6], preventing inherent-constant memory from being overwritten. When $\texttt{const}(a\downarrow^k$

$) = 0$ and $\texttt{obliv}(a \downarrow^{k'}) \neq \texttt{obliv}(b \downarrow^{k'})$ for some $k' \geq k$, assigning an oblivious value to $a \downarrow^{k'}$ would also assign it to the non-oblivious $b \downarrow^{k'}$, violating the dataflow theorem.

To satisfy Theorem 4, we have:

Rule 7 (Arithmetic) For an arithmetic operator expression, if any operands are oblivious, the total expression is also oblivious. Note that we only consider arithmetic expressions; comma expressions, compilation-time constant expressions (e.g., `sizeof`, `alignof`), and pointer reference/dereference operators are not considered dependencies.

Rule 8 (Dereference) It is forbidden to dereference an oblivious pointer (the array indexing expression `p[i]` is interpreted as `*(p + i)`). The result of a dereference operator shifts the `olevel` right by one digit, while the result of a take-address operator shifts the `olevel` left by one digit.

Rule 9 (Pointer Arithmetic) For the arithmetic of pointers, the only allowed type is the addition of a pointer `p` and an integer `i` (including the equivalent `&p[i]`). The 0th digit of the result `olevel` is the logical-OR of the operands' obliviousnesses, and the remaining digits are the same as `p`.

These rules ensure the theorems stated earlier. Note that Rules 6 to 8 replace their "basic" versions (Rules 1 to 3). The following example demonstrates how oblivious assignment rules work:

Example 3.

```
// [[obliv]] is the abbreviation of [[obliv("1")]]
[[obliv]] int x = 1;
[[obliv("10")]] int *xp = &x;
[[obliv("100")]] int **xpp = &xp;
// error, 2nd digit of olevel differs
[[obliv("110")]] int **ypp = xpp;
// ok, 2nd level is constant
[[obliv("110")]] int *const *const ypp = xpp;
// ok, the first level is passed by value
[[obliv("101")]] int **ypp = xpp;
```

3.3 Function Calls

Since callers typically only see function declarations in headers while their implications could exist in other translation units, obliviousness must be expressed in function signatures. We interpret function calls as series of assignments:

Rule 10 (Function Call) When checking a function, the following rules should be enforced:

1. Each parameter of a function can be annotated with an `olevel`.
2. When checking the function body, the parameters are treated as variables with their annotated `olevel`s.

3. The function itself can be annotated with an `olevel`, which indicates the `olevel` of its return value.
4. When checking the function body, every expression returned in a `return` statement should be able to be cast to the annotated `olevel` of the function, according to Rule 6.
5. When calling a function, for each parameter, the given argument should be able to be cast to the `olevel` of the declared parameter, according to Rule 6.

3.4 Classes

C++ classes (`struct`, `union`) declare compound types with methods, inheritance, and polymorphism, requiring additional rules:

Rule 11 (Class) For a class type (`struct` and `union` types also included), the following rules are enforced:

1. In the declaration of a class type, its members can be annotated with an `olevel`. The difference from a normal `olevel` declaration is that the last digit of class member `olevel` can be `"x"` besides `0` and `1`. If not specified, a class member has an `"x"` `olevel`.
2. In a class member access expression (e.g., `c.member` or `c->member`), if the `olevel` of the member declaration ends with `"x"`, the `olevel` of the expression is the `olevel` of the member declaration with the last digit changed to the `olevel` of the class instance (`c` or `*c`). Otherwise the `olevel` of the expression is the same as the member declaration.
3. A method can be annotated with `obliv_this`. In the presence of this annotation, `this` is interpreted as a pointer to an oblivious instance. An oblivious instance can only call `obliv_this` methods. A public instance can call all methods.
4. When a method is overridden during inheritance, the `olevel` annotations should be identical to the annotations in the original declaration.
5. Virtual methods should not be called from a pointer to an oblivious polymorphic pointer.

By default, a member's obliviousness inherits from the instance (`olevel` ending with `"x"`), analogous to C++'s `const`/`mutable` distinction. Virtual calls on oblivious polymorphic pointers are forbidden as the VPTR would be oblivious.

3.5 Templates

A single template could generate multiple instantiations. Since template parameters may carry `olevel` information (via function return values or parameters), obliv-clang checks each instantiation rather than the template declaration itself. This is straightforward as `clang` generates separate AST for each instantiation.

3.6 Unsafe Code Blocks

We provide the `obliv_unsafe` attribute to mark code blocks as unsafe, bypassing obliviousness checks. This approach, inspired by Rust's `unsafe` keyword, is necessary in several cases:

- Inspecting internal secret values when debugging;
- Implementing subroutines whose obliviousness requires sophisticated mathematics beyond automatic checking (e.g., extracting public values from oblivious variables through encryption/hashing with secret keys, or asserting on execution paths that should not trigger);
- Working with third-party libraries whose implementations are in separate translation units invisible to the compiler.

Like Rust's unsafe blocks, this feature indicates areas requiring special programmer attention and human auditing. We anticipate that in the future, more expressive semantics will reduce the need for unsafe blocks.

3.7 Known Limitations

When using external libraries, the compiler cannot determine the obliviousness of symbols without `olevel` annotations in library headers. Programmers must add these annotations, though this is typically a moderate burden. Another limitation involves function overloading. Since `olevel` is not part of C++ types, overloading by `olevel` alone is prohibited unless encoded in the ABI.

4 Implementation

obliv-clang is implemented in approximately 800 lines of C++ code. It registers an annotation handler to parse `obliv`-related annotations and a frontend plugin that executes after `clang` parses the source code into an AST. The plugin scans the AST and raises compilation errors for code violating our rules.

obliv-clang works like other `clang` plugins: it compiles into a dynamic library loaded during compilation, requiring only one additional compiler option to load the plugin. Our implementation targets a wide range of LLVM monorepo versions (tested on both LLVM 13 and LLVM 20). `clang`'s AST data structure remains relatively stable, so minimal maintenance is needed to support newer versions.

5 Compiler Optimization Suppression

Modern compilers perform aggressive performance optimizations that reorganize program structures. While obliv-clang ensures obliviousness at the AST level, the LLVM optimizer may apply transformations that undermine these guarantees.

Consider this example from [26]:

```
// Select an element from an array in constant time
uint64_t constant_time_lookup(
        const size_t secret_idx, const uint64_t table[16]) {
    uint64_t result = 0;
    for (size_t i = 0; i < 16; i++) {
        const bool cond = i == secret_idx;
        const uint64_t mask = (-(int64_t)cond);
        result |= table[i] & mask;
    }
    return result;
}
```

This function uses bit operations to extract `table[secret_idx]`, without revealing `secret_idx` through side channels. However, LLVM's optimizer can detect this pattern and bypass the protection. When compiled with `clang` 21.1.0 targeting x86-64 Linux using `-O3 -fno-unroll-loops -fno-vectorize`, the compiler generates a branch depending on whether `i` equals to `secret_idx`, allowing attackers to learn information about `secret_idx` by observing control flow, violating the intended obliviousness.

This issue arises because current compiler optimizers prioritize correctness without considering obliviousness. Reconstructing LLVM to be obliviousness-aware would require enormous engineering effort. Instead, we adopt a simpler approach. When our plugin loads, it disables optimization passes that could produce unsafe changes. We manually inspected all `clang` optimization passes to determine their safety. Here is the list: ReassociatePass, LoopInstSimplifyPass, LoopSimplifyCFGPass, LoopRotatePass, IndVarSimplifyPass, LoopDeletionPass, LoopInterchangePass, LoopFlattenPass, X86CmovConverterPass.

To ensure the completeness of necessary passes to disable, obliv-clang provides an extra double-check step that applies rigorous obliviousness verification to the generated assembly code using existing tools [8]. The performance impact of disabling these optimization passes is insignificant, as evaluated in Section 6.2.

6 Evaluation

We evaluate obliv-clang's expressiveness through case studies, and assess its performance through compilation and runtime benchmarks.

6.1 Case Studies

PathORAM PathORAM [28] provides an oblivious random access interface with $O(\log^3 N)$ overhead. Our non-recursive implementation is 251 lines of code (LOCs). The position map requires `obliv_unsafe` blocks since positions read from the secret map must become public for tree traversals.

Oblivious BFS Oblivious BFS [3] performs BFS traversals such that an attacker cannot learn the graph structure from the execution trace. It shuffles the graph randomly and chooses each next vertex with equal probability. Our implementation uses 158 LOCs.

Real-World Cryptographic Vulnerabilities CVE-2024-13176 (the Minerva attack) is a timing side channel in OpenSSL's ECDSA: a ∼300 ns signal

leaks when the inverted nonce's top word is zero, enabling private key recovery on NIST P-521. The root cause was that `BN_mod_exp_mont()` lacked constant-time protection for its secret base and result, despite protecting the exponent. With obliv-clang, this flaw is caught automatically:

```
int BN_mod_exp_mont(BIGNUM *r, const BIGNUM *a, const BIGNUM *p,
                    const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx);
static int ec_field_inverse_mod_ord(const EC_GROUP *group,
        OBLIV_PTR BIGNUM *r, OBLIV_PTR const BIGNUM *x, BN_CTX *ctx) {
    if (!BN_mod_exp_mont(r, x, ...))  // ERROR: OBLIV_PTR to non-OBLIV_PTR
        goto err;
}
```

obliv-clang's explicit dataflow tracking catches secret values flowing into unannotated functions, demonstrating robustness over naming conventions alone.

Others We have also tested obliv-clang on several other algorithms, including oblivious shuffle [18] and tree-based data structures [32]. We also write some standard library wrappers to assist in using standard libraries. The adoption of these programs is straightforward. The only requirement is to mark variables and function parameters with appropriate `olevels`.

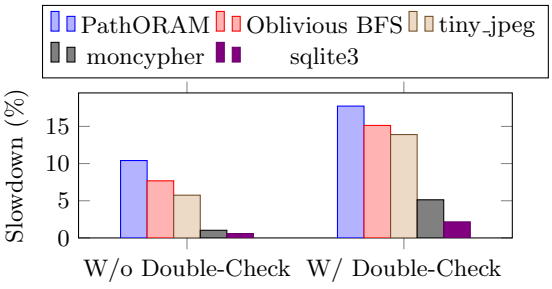
To quantify the migration effort, we measure the LOCs requiring `[[obliv]]` annotations compared to the total LOCs. For example, we annotated 32 out of 190 LOCs in PathORAM, 10 out of 94 LOCs in Oblivious BFS, and 25 out of 311 LOCs in Poly1305 (see later). This demonstrates that the annotation burden is modest — typically just 8% to 17% of the codebase requires modifications, and these modifications are localized to variable declarations and function signatures involving secret data.

6.2 Performance

All experiments run an Intel Xeon Gold 5218R processor with 256 GB DDR4 memory, with Ubuntu 22.04. Programs are compiled with `clang` 13.0.1 at `-O3`; with obliv-clang, optimization passes mentioned in Section 5 are disabled.

Compilation Overheads We evaluate compilation overheads on five benchmarks: PathORAM (251 LOCs, Section 6.1), Oblivious BFS (158 LOCs, Section 6.1), tiny_jpeg (1306 LOCs, JPEG encoding) [9], monocypher (3277 LOCs, cryptography library) [30], and sqlite3 (250815 LOCs) [27]. All programs are compiled (but not linked) with and without our obliv-clang plugin. Results are shown in Fig. 1a (average of 10 runs after 3 warm-ups). Without the double-check step (Section 5), overheads are below 11%; with it, below 18%. Overheads decrease for larger programs, making obliv-clang suitable for real-world codebases.

Optimization Suppression Overheads We measure execution time with and without suppression on: PathORAM with randomly generated operations on size 2^{20}, Oblivious BFS on a 4096-node graph, tiny_jpeg encoding a 1000×1000 image, and lz4 [17] with compression level 3. Fig. 1b shows the results. Most benchmarks see negligible slowdown, with lz4 even slightly faster. PathORAM slows down by 29.5%, likely because the suppressed passes could reorganize its loops for better performance.



(a) Compilation time overheads.

Testcase	Slowdown
PathORAM	29.5%
Oblivious BFS	0.91%
tiny_jpeg	1.27%
lz4	−2.72%

(b) Optimization suppression overheads.

Fig. 1: Overheads from compilation process and optimization suppression.

Runtime Performance We compare obliv-clang against FaCT [5] (a DSL that verifies and transforms oblivious code) and Jasmin [1] (a DSL with formally verified compilation) on six algorithms: AES128, poly1305, salsa20, secretbox, siphash [2], and PathORAM. Both FaCT and Jasmin lack usability features (e.g., limited loop constructs, manual register allocation), restricting comparison to simple microbenchmarks. The results are shown in Fig. 2. obliv-clang is 27.8% faster than Jasmin and 66.9% faster than FaCT on average. For poly1305, the obliv-clang implementation is 29.8% slower than Jasmin. The reason might be that the main body of poly1305 calculation is 128-bit integer arithmetic, and the code generation of 128-bit arithmetic in Jasmin is more optimized.

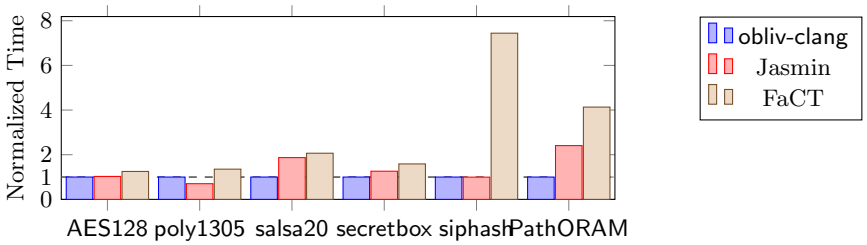


Fig. 2: Performance comparison of oblivious programs using different tools.

7 Conclusion

We present obliv-clang, a compiler plugin for the C++ language that performs static checks for oblivious programming. It is specifically implemented using the `clang` and LLVM infrastructures. Compared to previous work, obliv-clang aims to provide natural semantics and maximum compatibility with widely used

industry-level programming languages and workflow while being able to provide a verifiable soundness guarantee even in the presence of complex language features. Our experiments show that obliv-clang leads to low compilation overheads and can achieve better performance on the compiled oblivious programs compared to previous tools.

Acknowledgments. The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. Mingyu Gao is the corresponding author. The technical solution and experimental design presented in this paper were independently completed by the authors. AI tools were used solely for language polishing and formatting optimization, and did not participate in the development of the research ideas or core content.

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-Assurance and High-Speed Cryptography. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1807–1823 (Oct 2017)
2. Aumasson, J.P., Bernstein, D.J.: SipHash: A Fast Short-Input PRF. In: Progress in Cryptology - INDOCRYPT 2012. pp. 489–508 (2012)
3. Blanton, M., Steele, A., Alisagari, M.: Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In: ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS). p. 207 (2013)
4. Bulck, J.V., Weichbrodt, N., Kapitza, R., Piessens, F., Strackx, R.: Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security Symposium. pp. 1041–1056 (Aug 2017)
5. Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R.S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., Stefan, D.: FaCT: A DSL for Timing-Sensitive Computation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 174–189 (Jun 2019)
6. Clang-Tidy — Extra Clang Tools 18.0.0git Documentation. https://clang.llvm.org/extra/clang-tidy/ (2023)
7. What is Clangd? https://clangd.llvm.org/ (2023)
8. Disselkoen, C., Cauligi, S., Tullsen, D., Stefan, D.: Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork. In: SRC TECHCON (2020)
9. Gonzalez, S.: TinyJPEG (Aug 2022)
10. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache Attacks on Intel SGX. In: European Workshop on Systems Security (EuroSec). pp. 1–6 (Apr 2017)
11. Jang, Y., Lee, J., Lee, S., Kim, T.: SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: Workshop on System Software for Trusted Execution (SysTEX). pp. 1–6 (Oct 2017)
12. Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. pp. 557–574 (2017)
13. Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., Gruss, D.: PLATYPUS: Software-based Power Side-Channel Attacks on X86. In: IEEE Symposium on Security and Privacy (S&P). pp. 355–371 (May 2021)

14. Liu, C., Hicks, M., Shi, E.: Memory Trace Oblivious Program Execution. In: IEEE Computer Security Foundations Symposium (CSF). pp. 51–65 (Jun 2013)
15. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: A Programming Framework for Secure Computation. In: IEEE Symposium on Security and Privacy (S&P). pp. 359–376 (May 2015)
16. Luo, Y., Gao, M.: obliv-clang: Real-World Oblivious Programming in C++ (Extended Version). arXiv preprint cs.PL/2606.16187 (2026), https://arxiv.org/abs/2606.16187
17. LZ4 - Extremely Fast Compression. https://lz4.org/ (2023)
18. Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The Melbourne Shuffle: Improving Oblivious Storage in the Cloud (Feb 2014)
19. Rane, A., Lin, C., Tiwari, M.: Raccoon: Closing Digital Side-Channels Through Obfuscated Execution. In: USENIX Security Symposium. pp. 431–446 (2015)
20. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is My Code Constant Time? (2016)
21. Sayakkara, A., Le-Khac, N.A., Scanlon, M.: A Survey of Electromagnetic Side-Channel Attacks and Discussion on Their Case-Progressing Potential for Digital Forensics. Digital Investigation **29**(1), 43–54 (Jun 2019)
22. Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In: Network and Distributed System Security Symposium (NDSS). Internet Society, San Diego, CA (2017)
23. Shinde, S., Chua, Z.L., Narayanan, V., Saxena, P.: Preventing Page Faults from Telling Your Secrets. In: ACM Asia Conference on Computer and Communications Security (ASIA CCS). pp. 317–328 (May 2016)
24. Sinha, R., Rajamani, S., Seshia, S.A.: A Compiler and Verifier for Page Access Oblivious Computation. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE). pp. 649–660 (Aug 2017)
25. Son, J., Prechter, G., Poddar, R., Popa, R.A., Sen, K.: ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State. In: USENIX Security Symposium (2021)
26. Sprenkels, D.: LLVM Provides No Side-Channel Resistance. https://dsprenkels.com/cmov-conversion.html (Oct 2019)
27. SQLite Home Page. https://www.sqlite.org/index.html (Oct 2023)
28. Stefanov, E., Dijk, M.V., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: An Extremely Simple Oblivious RAM Protocol. Journal of the ACM (JACM) **65**(4), 1–26 (2018)
29. The C++ Standards Committee: C++ Standard Draft Sources. https://github.com/cplusplus/draft (Oct 2022)
30. Vaillant, L.: Monocypher. https://monocypher.org/ (2023)
31. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. pp. 991–1008 (Aug 2018)
32. Wang, X.S., Nayak, K., Liu, C., Chan, T.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious Data Structures. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 215–226 (2014)
33. Zahur, S., Evans, D.: Obliv-C: A Language for Extensible Data-Oblivious Computation (2015)
34. Zhang, Z., Barthe, G.: CT-LLVM: Automatic Large-Scale Constant-Time Analysis. Cryptology ePrint Archive, Report 2025/338 (2025)