


CT-LLVM: Automatic Large-Scale Constant-Time Analysis

Zhiyuan Zhang , Gilles Barthe  

 *MPI-SP, Bochum, Germany*

 *IMDEA Software Institute, Madrid, Spain*

Abstract

Constant-time (CT) is a popular programming discipline to protect cryptographic libraries against micro-architectural timing attacks. One appeal of the CT discipline lies in its conceptual simplicity: a program is CT iff it has no secret-dependent data-flow, control-flow or variable-timing operation. Thanks to its simplicity, the CT discipline is supported by dozens of analysis tools. However, a recent user study demonstrates that these tools are seldom used due to poor usability and maintainability (Jancar et al. IEEE SP 2022).

In this paper, we introduce CT-LLVM, a CT analysis tool designed for usability, maintainability and automatic large-scale analysis. Concretely, CT-LLVM is packaged as a LLVM plugin and is built as a thin layer on top of two standard LLVM analysis: def-use and alias analysis. Besides confirming known CT violations, we demonstrate the usability and scalability of CT-LLVM by automatically analyzing nine cryptographic libraries. On average, CT-LLVM can automatically and soundly analyze 36% of the functions in these libraries, proving that 61% of them are CT. In addition, the large-scale automatic analysis also reveals new vulnerabilities in these libraries. In the end, we demonstrate that CT-LLVM helps systematically mitigate compiler-introduced CT violations, which has been a long-standing issue in CT analysis.

1 Introduction

The constant-time (CT) discipline is commonly used to protect cryptographic libraries against micro-architectural timing attacks. Informally, the CT discipline mandates that programs should not have secret-dependent data-flow, control-flow, or variable-timing operations. While the CT discipline is conceptually simple, it remains a challenge for developers to write efficient CT code. As an attempt to mitigate the issue, many tools have been developed over the last fifteen years to help developers writing CT code. Yet, despite the availability of these tools, CT violations are still consistently found in cryptographic libraries.

Problems Identification. We identify two main reasons for not closing the gap between the CT discipline and the practice. The first reason is the low adoption of CT analysis tools in real-world development. A recent study [24] shows that developers do not routinely use CT analysis tools because of poor usability. First, most available tools are difficult to install, due to complex dependencies and reliance on deprecated software. Second, once installed, the overwhelming majority of the tools are still hard to use. For instance, they may require complex setups for each use of the tool. Third, analysis results may be difficult to interpret, due to the underlying analysis techniques, or to the lack of clear mechanisms to locate the root cause of leakages.

The second reason is that current tools are designed with expectation that the user needs to choose a specific function/execution path to analyze, and to annotate the secrets [24]. Consequently, the user can only analyze a small portion of the codebase, such as the parts they are familiar with or suspect to be CT insecure, leaving large fragments of the libraries outside of their analysis. As an example, it was recently discovered that Base64 decoding is not CT in most cryptographic libraries[35]. Base64 decoding does not participate in cryptographic operations directly, but is necessary to convert asymmetric keys from PEM format to binary format. Surprisingly, the implementation of Base64 decoding is as simple as looking up a predefined table. However, due to the lack of exposure to cryptographers, the function has not been analyzed by any CT analysis tools until 2021.

Contributions. While we can keep relying on occasional studies to find CT violations, we believe that a usable tool aiming for automatic large-scale sound analysis is necessary to make cryptographic libraries CT. In this paper, we design, implement and evaluate CT-LLVM, a new CT analysis tool for LLVM programs, aiming to address the aforementioned problems. The tool has five main features:

- **usability:** CT-LLVM is implemented as a LLVM plugin and is thus easy to install, to maintain and to run. CT-LLVM delegates the core of the analysis to the LLVM; specifically, CT-LLVM relies on LLVM for alias and

data-dependency analysis. As a result, the code of the plugin is compact, and easy to maintain. Furthermore, as a LLVM plugin, the tool can be used seamlessly by library developers when compiling the project;

- **transparency:** CT-LLVM’s analysis operates directly on LLVM IR, avoiding potentially problematic translations to alternative representations—such translations are typically not designed for security analysis and may potentially introduce or remove CT violations [30];
- **soundness:** CT-LLVM tracks the propagation of secrets and detects their flows into memory address, branches and variable-timing operations. The analysis is sound when the tool captures the entire propagation of secrets.
- **automation:** CT-LLVM supports automatically analyzing the entire library, without specifying the functions and variables to analyze. To be specific, CT-LLVM, as a static information-flow analysis tool, can soundly analyze functions that do not have external function calls (after inlining), function pointers, and inline assembly.
- **customization:** CT-LLVM is implemented in a modular way and has a simple CT analysis logic. Therefore, it is easy to customize and extend the tool for specific needs. For example, the tool can be used to mitigate CT violations introduced by compiler optimizations.

To summarize, we make following contributions in this paper:

- We propose and implement a CT analysis tool which only relies on the LLVM infrastructure;
- We build a database consists of vulnerable cryptographic implementations and demonstrate that our tool achieves the same or better performance, compared to other tools;
- We demonstrate that our tool can be used to verify the absence of CT violations with examples;
- We automatically find CT violations and prove their absence in nine popular cryptographic libraries;
- We demonstrate that our tool can be used to mitigate CT violations introduced by compiler optimizations. Interestingly, we show that fixes for CT violations lead to new vulnerabilities.

Outline. The rest of the paper is organized as follows. We first introduce the background and related work of CT analysis tools in Section 2. Then we explain the design choices of our tool in Section 3, and LLVM features that we leverage in Section 4. After that, we present the implementation of our tool in Section 5. Our evaluation comes in three parts: we confirm known CT violations in Section 6; we detail the procedure to verify the absence of CT violations with an example of Kyber in Section 7; and we perform a large-scale evaluation in Section 8. In the end, we demonstrate that CT-LLVM detects CT violations introduced by compiler optimizations in Section 9. We conclude the paper in Section 10 and detail the results and new CT violations from disclosing our findings to the studied cryptographic libraries in Section 11.

2 Background & Related Work

In this section, we first introduce the background of micro-architectural timing attacks and the CT policy. Then, we present related work on building and using CT analysis tools.

Micro-architectural Timing Attacks. Program execution is affected by the availability of micro-architectural resources, such as the cache and the branch predictor. Micro-architectural timing attacks either actively affect the victim program’s execution or passively monitors its execution. Often, these attacks exploit secret-dependent cache accesses [27, 44, 45], and control flow [1, 20, 46] to infer the value of the secret. Besides, variable-time instructions, such as division, can also be exploited to leak secrets [4, 11].

CT Policy. To prevent from having variable execution time with different secrets, constant-time (CT) policy requires that the secret should not affect the memory access and the control flow. A strict CT policy, which is the one we follow in this paper, further requires that the secret should not be processed by variable-time instructions.

Classification of CT Tools. Barbosa et al. [8] and Geimer et al. [22] have classified the CT analysis tools based on their formal guarantees and the type of analysis they perform, respectively. Existing CT analysis tools can be classified into three categories. First, dynamic analysis collects execution traces and analyze whether different inputs cause different memory access and control flow [40, 41, 42]. Second, statistical analysis [32] collect the actual execution time of the program and statistically analyze the timing variance. Third, static analysis leverages symbolic execution [16], abstract interpretation [17, 18], information-flow analysis [33, 47] or logical reduction [2] to provide different-level of formal guarantees. In contrast, the methodologies of dynamic and statistical analysis cannot provide soundness guarantees.

Usability of CT Analysis Tools. Jancar et al. [24] and Fourné et al. [21] study the usability of CT analysis tools and their deployment in the development process. Jancar et al. [24] shows that installing the tools is painful and the cost of setting up a test, such as writing a test wrapper, may cost more time than just browsing the code. Fourné et al. [21] shows that the tools are not routinely used by developers. Only five tools out of 49 surveyed tools are used by developers in 15 out of 27 surveyed cryptographic libraries. Particularly, 11 of these 15 libraries use the same tool, ctgrind [26], which is a dynamic analysis tool that patches the valgrind.

LLVM-based CT Analysis Tools. There exist several tools operate on LLVM IR [2, 6, 14, 33, 47] that provide different levels of formal guarantees by leveraging logical reduction or information-flow analysis. From the view of usability, they require the source code to be first compiled to IR code before the analysis which makes it hard to be integrated into existing projects. Furthermore, some tools rely on deprecated formal verifiers [2, 6, 14] or specific versions of LLVM [33, 47], which makes it hard to maintain and scale the analysis.

Limitation of Analyzing IR Code. CT analysis that operates on IR code has been long criticized for not capturing leakages caused by compiler optimizations. Examples of such leakages have been separately reported or discussed in various works [8, 16, 21, 22, 37]. Recently, Schneider et al. [34] carries out a systematic study with binary-level dynamic CT analysis and show that both LLVM and GCC optimize CT code into conditional branches under three scenarios. Particularly, LLVM introduces leakages by lowering the `select` instruction into conditional branches. Such optimization happens after LLVM issues the IR code that is analyzed by current LLVM-based CT analysis tools. Therefore, the leakages cannot be captured by these tools.

3 Design Goals and Choices

We propose CT-LLVM with the goal of providing a usable tool that can automatically detect CT violations and prove their absence in cryptographic libraries. In this section, we discuss the criteria that we consider important for a CT analysis tool, and our trade-offs in designing CT-LLVM.

3.1 Sound Analysis Methodology

First of all, a CT analysis tool should have a sound analysis methodology. That is, when the CT analysis tool reports that the test target has no CT violations, it should be guaranteed that the test target is indeed CT. Compared to just detecting CT violations, proving their absence is more challenging. In fact, according to a recent study [21], only 16 out of 49 CT analysis tools have a sound methodology.

Our approach. We consider CT analysis as a taint analysis problem. Intuitively, the analysis tracks the propagation of a secret input and check whether a tainted instruction breaches the CT policy. If the taint analysis captures all instructions that depend on the secret input, it inherently can detect all CT violations, assuring the soundness of the analysis. Furthermore, a sound CT analysis does not need to consider CT violations caused by implicit information-flow, as a program is already not CT if it branches on a secret. Fixing a CT violation caused by explicit information-flow automatically fixes the implicit information-flow that depends on it.

We notice that capturing explicit information-flow of a secret input is naturally solved by compilers, which analyze the dependency between instructions to perform optimizations. For example LLVM leverages def-use chains to track the propagation of static single assignment (SSA) variables and alias analysis to track the propagation of memory objects. Therefore, we simply leverage these two analysis to track the propagation of secret in CT-LLVM.

Trade-offs. While the methodology is straightforward in theory, it has limitations in practice. First, compiler treats inline assembly as a black box. Therefore, the analysis cannot reason whether an inline assembly breaches the CT policy or

whether it affects memory objects that depend on the secret input. Second, compiler cannot reason about functions that are only declared, but not defined in the same translation unit. Similarly, the analysis cannot reason indirect branches, including indirect function calls, as the target of the branch is not known at compile time. These limitations can be addressed by over-approximating the analysis and providing function signatures that define the behavior of declared functions. However, the former approach often comes with high false positives, while the latter approach requires manual effort to provide function signatures, which makes the tool less usable.

Our Implementation. We choose to accept these limitations in CT-LLVM and focus on providing a tool that is easy to use and has low false positives. To be specific, we configure CT-LLVM to provide two modes of analysis: *Proof Mode* and *Violation Finding Mode*. Proof Mode enforces a sound analysis and do not analyze functions that cannot be fully inlined. Violation Finding Mode assumes non-inlined functions are CT and does not modify memory content with secret-dependent values. Therefore, this mode can find CT violations but cannot prove their absence. The intuition of having a Violation Finding Mode is to cover functions that cannot be analyzed in proof mode.

3.2 Platform-(In)dependent Analysis

Since def-use chains and alias analysis are LLVM analysis passes for LLVM IR programs, CT-LLVM thus operates on LLVM IR programs, which are platform-independent. The advantage of analyzing LLVM IR programs is that it is a cheap and effective way to analyze platform-dependent programs by cross-compiling them to LLVM IR on any host platform.

Platform-dependent Leakages. One potential concern with analyzing LLVM IR programs is that it is believed to miss compiler-induced CT leakages [16, 19, 37, 43], which are platform-dependent. Such optimizations often come at the backend of the compiler, where the compiler generates machine code from the IR. Simon et al. [37] and Daniel et al. [16] show that a source code can be CT on x86-64, but not on i386. Schneider et al. [34] summarizes three classes of optimizations that introduce conditional branches on secret-dependent values under different optimization flags and for different target platforms. Interestingly, we find that all three classes of introduced leakages are caused by the `select` instruction in LLVM IR, which is used to select a value based on a condition. That is, currently identified LLVM-induced CT leakages are all connected to this special instruction.

Our approach. Since the `select` instruction is introduced at LLVM IR level, CT-LLVM can thus naturally capture the existence of them. This is done by adding one more rule to the CT policy: a `select` instruction breaches the CT policy if its condition is secret-dependent. Compared to the dynamic analysis approach used by Schneider et al. [34], our approach

is much cheaper and can detect all such leakages just by compiling the program with CT-LLVM.

Trade-offs. Our approach over-approximates that all `select` instructions will be converted to conditional branches in the machine code. Due to the complexity of the backend optimization, it is unclear under which combination of optimization flags and target platforms, the `select` instruction will be lowered to conditional branches. However, we find that starting from LLVM18, if the `select` instruction is lowered to a conditional move, it can always be converted to a conditional branch with flag `x86-cmov-converter-force-all`. This indicates that with the progress of LLVM, new flags can be added to control the optimization of `select` instructions. Therefore, we argue that only completely removing the `select` instruction from the IR can prevent such leakages.

3.3 Usability and Maintenance

So far, we have discussed two important criteria for CT analysis: soundness and detecting both platform-independent and dependent CT violations. We now discuss how we make CT-LLVM usable and maintainable.

Minimize False Positives. CT-LLVM minimizes false positives by performing flow-sensitive and context-sensitive taint analysis to track the propagation of secrets. Flow-sensitive analysis is achieved by the nature of def-use chains and the reachability analysis implemented in LLVM for alias analysis. Context-sensitive analysis is achieved by inlining functions with an API provided by LLVM. We note that neither of two sound LLVM-based CT analysis tools proposed in recent years, Flowtracker [2] and CT-Checker [33], supports both flow-sensitive and context-sensitive analysis at the same time. Specifically, Flowtracker does not reason about invoked functions and is flow-insensitive, while CT-Checker is flow-insensitive but context-sensitive.

We adopt a type system to further filter false positives according to a common assumption in CT threat model: addresses, unless tainted, are considered public. In fact, the only source of false positives in CT-LLVM comes from the alias analysis, which is conservative, implemented in LLVM. However, we argue that the false positives caused by alias analysis can possibly be reduced with the improvement of LLVM.

Usability. We implement CT-LLVM as a LLVM plugin, which can be easily integrated into the compilation process. Specifically, it can be integrated into GitHub Continuous Integration (CI), as GitHub CI has LLVM pre-installed. CT-LLVM supports two ways to use. First, it can be invoked when compiling the program. The user only needs to pass a flag `-fpass-plugin=ctlvm.so` to `clang` to enable CT-LLVM. Compared to other tools that analyze LLVM IR programs [2, 33, 47], CT-LLVM does not require the user to first convert the program to an alternative form, such as `.bc` or `.ll` files. Our approach largely ease the integration of CT-LLVM into the development pipeline of cryptographic libraries.

Second, CT-LLVM can be used as a standalone tool to analyze LLVM IR programs “decompiled” from binaries. This is achieved by keeping IR instructions in the binary when compiling the program with `-fembed-bitcode` flag. This approach also does not require changes to the compilation process, and it benefits the sound analysis by putting all function implementations in the same file. Therefore, more functions can be inlined for sound analysis.

Annotation. CT-LLVM does not force the user to annotate the program to specify secrets. Instead, it can automatically analyze all function arguments as if they are secret. After the analysis, CT-LLVM generates a report for each tainted function argument, reporting whether it breaches the CT policy at which line of code. The user only needs to identify whether arguments that breach the CT policy are secret or not. In contrast, it does not matter whether an argument may hold a secret when it never breaches the CT policy.

Maintenance. The core logic of tracking the propagation of secrets leverages functionalities provided by LLVM. Therefore, the maintenance of CT-LLVM is partially supported by the open-source community of LLVM. On the other hand, CT-LLVM serves as a glue to stick these functionalities together for CT analysis. This makes the codebase of CT-LLVM simple and compact, which also reduces the maintenance cost.

3.4 Transparency

In the end, we discuss the transparency of CT-LLVM. The concept is first introduced by a recent study [30], where *transparency* means that the transformation of a program to an alternative form does not introduce or eliminate CT violations. Transparency is essential for the soundness of the tool. For example, binary lifter may optimize the lifted program, just like a compiler, to improve the readability. This may result in eliminating CT violations (e.g., dead load) in the lifted program. Similarly, Zhou et al. [47] reports that lifter used by CacheS [39] lifts conditional moves into conditional branches, which introduce CT violations that do not exist in the binary. Such incorrect lifting introduces new CT violations.

Our approach. The transformation of source code to LLVM IR is transparent if all optimizations are disabled. Hence, using `-O0` helps guarantee the transparency of the transformation. Besides the default transformation done by LLVM, CT-LLVM further leverages the `mem2reg` promotion and function inlining to improve the precision of the analysis. The former transformation promotes stack variables that temporarily hold the value of a secret to SSA variables, while the latter transformation replaces function calls with the function body. These two transformations are also transparent, as they do not introduce or eliminate CT violations.

Trade-offs. CT-LLVM operates under all optimization levels to make the tool usable. Since optimizations may remove codes, such as dead loads or branches, it is possible that CT-LLVM may not be transparent under optimization levels other

than `-00`. We make this trade-off to make CT-LLVM usable, and state optimizations affect the transparency of CT-LLVM.

3.5 Summary of Design Choices

In summary, CT-LLVM is a flow-sensitive and context-sensitive CT analysis tool that supports transparent transformation of source code to the interpretation analyzed by the tool. It has two modes to prove the absence of CT violations and detect CT violations, respectively. Although CT-LLVM analyzes platform-independent LLVM IR programs, it supports detecting platform-dependent CT violations that are introduced by the `select` instruction. Since CT-LLVM is implemented as a LLVM plugin and leverages functionalities provided by LLVM, it can be easily integrated into existing development pipelines and has a low maintenance cost.

4 LLVM Features for CT Analysis

In this section, we present LLVM features that supports our sound analysis mechanism. We begin by introducing the static single assignment (SSA) form, which is the foundation for compiler analysis and optimization. Next, we delve into LLVM’s implementation of the def-use chain and alias analysis, highlighting how they contribute to the CT analysis. In the end, we describe how we make alias analysis flow-sensitive with reachability analysis.

4.1 LLVM’s Static Single Assignment

Static single assignment (SSA) assigns each SSA value only once and creates a new SSA value whenever it gets reassigned. Presenting the code in SSA form largely simplifies the dependency analysis and thus is widely used in compilers.

SSA Form in LLVM. LLVM also structures its intermediate representation (IR) in SSA form. However, by default, it avoids introducing multiple SSA values for the same variable by always spilling SSA values to the stack. Therefore, the SSA value is loaded from the stack whenever needed and stored back after it is modified. Data flow analysis does not benefit from such a semi-SSA form, as the data propagation is not explicitly represented in SSA form. To convert such IR into a strict SSA form, LLVM provides the `mem2reg` pass, which promotes stack operations to SSA values. Specifically, this pass replaces pairs of load and store operations with a single SSA value. Our analysis is conducted on this strict SSA form, obtained by applying the `mem2reg` pass.

Example. To illustrate the transformation from source code to strict SSA form, we present a simple C function and its corresponding LLVM IR in [Figure 1](#). Later, we reuse this example for demonstrating LLVM’s data-dependency analysis. The function takes a public value `pub` and a secret value `sec` as inputs. An internal variable, `tmp`, is first assigned the value of `sec` incremented by one. Then this value is assigned a new

value depends on the value of `pub`. In the end, the function checks if `tmp` is non-zero. This operation may or may not leak the secret value, depending on the value of `pub`.

The IR in strict SSA form is shown on the right side of [Figure 1](#). We highlight the SSA variable that represents `tmp` in red. As we can see from the IR, different SSA values are used to represent the same variable at different points in the program. Particularly, the conditional update of `tmp` based on `pub` is represented by a `phi` instruction, which selects the value of `tmp` based on the predecessor block. With the `phi` instruction, the compiler easily knows that `tmp` can potentially hold zero or `sec` based on the value of `pub`.

1 <code>func(pub, sec) {</code>	1 define <code>@func(%0, %1) {</code>
2 <code> tmp = sec + 1;</code>	2 <code> %2 = add %1, 1</code>
3 <code> if (pub)</code>	3 <code> %3 = icmp ne %0, 0</code>
4 <code> tmp = 0;</code>	4 <code> br i1 %3, label1, label2</code>
5 <code> else</code>	5 <code> label1: br label3</code>
6 <code> tmp = sec;</code>	6 <code> label2: br label3</code>
7 <code> if (tmp) { ; }</code>	7 <code> label3: %4 = phi [0, label1],</code>
8 <code> }</code>	8 <code> [%1, label2]</code>
9 <code> }</code>	9 <code> %5 = icmp ne %4, 0</code>
10 <code> }</code>	10 <code> br i1 %5, label4, label5</code>
11 <code>}</code>	11 <code>}</code>

C Code

LLVM IR

Figure 1: An example LLVM IR in strict SSA form

4.2 LLVM’s Def-Use Chain

Based on the SSA form, LLVM maintains a def-use chain for each SSA variable, which records the direct use of the variable. In other words, the chain does not track the indirect use of a variable. For example, the def-use chain of `x` contains `p := x + 1` but do not contain the use of `p`. However, we can easily extend the def-use chain by recursively querying the def-use chain of the uses (e.g., query the use of `p` in this case).

Def-Use Chain for CT Analysis. We leverage the def-use chain to track the propagation of secrets, which we refer to as *taint source*. Particularly, we taint a secret value and then recursively query the def-use chain to find all operations that depend on the secret. Once we have a complete list of the propagation of the secret, we can search for CT violation.

Soundness Discussion. Since the def-use chain operates on the SSA form, it can capture the entire propagation of a secret through SSA values. Therefore, the CT analysis based on the def-use chain is sound when the secret is never stored to the memory. Once the secret has been stored to the memory, the def-use chain loses the track of the secret as it cannot reason whether a variable is loaded from the same or aliased location that stores the secret.

Application Demo. We present a demo of traversing the def-use chain with the IR example in [Figure 1](#). The analysis

starts by tainting the secret value `sec` and querying the def-use chain of `%1`, which is the SSA variable of `sec`. This query returns the uses of it at line 2 and line 7. Both line 2 and line 7 define new SSA values, `%2` and `%4`, respectively. Therefore, we query the def-use chain of `%2` and `%4`, and find that `%4` defines a new SSA variable `%5` at line 8. Finally, we query the def-use chain of `%5` and find that it is used in the branch instruction at line 9. We visualize the def-use chain of `%1` in **Figure 2**. By recursively querying the def-use chain, we are able to build a list that captures the entire propagation of the secret. With a naive violation detection rule, which marks a CT violation whenever a branch is found in the list, we can conclude that the function violates the CT policy.

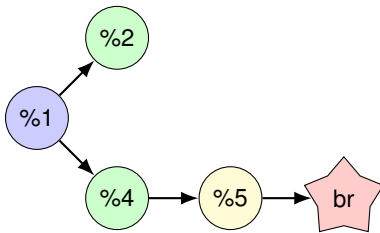


Figure 2: A visualized def-use chain of `sec`.

4.3 LLVM’s Alias Analysis

Alias analysis determines whether two pointers point to the same memory location. This feature perfectly addresses the limitation of the def-use chain, which cannot determine whether the address of a load operation is aliased with the address of a store operation.

Alias Analysis for CT Analysis. We use alias analysis to track the propagation of a secret across memory operations. Specifically, we query the alias analysis API provided by LLVM with the addresses of a store and a load operation. We add the loaded value to the taint list if the API suggests that the two addresses are aliased. Then, we query the def-use chain of the loaded value to enrich the taint list.

Soundness Discussion. With alias analysis, we can track the propagation of a secret across memory operations. By incorporating alias analysis with the def-use chain, we are able to capture the entire propagation of a secret through both SSA variables and memory operations. Therefore, CT analysis based on the def-use chain and alias analysis is sound.

LLVM’s Alias Analysis. LLVM implements multiple alias analysis algorithms, however most of them are implemented in an external analysis pass [28], and have long been deprecated¹. We leverage the core API provided by the release version of LLVM, called `BasicAA`, for alias analysis.

The alias analysis we leverage is field-sensitive, but not context-sensitive or flow-sensitive. Furthermore, the analysis is conservative to ensure the correctness of the analysis.

¹<https://github.com/seccuresystemslib/poolalloc>

Therefore, LLVM’s alias analysis may over-approximate the alias relationship between two pointers, which may introduce false positives. For each query, it returns four possible outcomes: *MustAlias*, *NoAlias*, *PartialAlias*, and *MayAlias*. The first two outcomes indicate that two pointers do or do not point to the same memory location. *PartialAlias* indicates that two pointers point to overlapping memory locations. *MayAlias* is returned when the analysis cannot determine the alias relationship between two pointers. Therefore, even two pointers are not aliased, the analysis may still return *MayAlias*.

Algorithm of Incorporating Alias Analysis. In a nutshell, the algorithm iterates each tainted store operation and checks whether the store address is aliased with any untainted load operation in the target function. Therefore, a nested loop is used to iterate all tainted store operations and untainted load operations. We store the results that LLVM can confirm in one list, and the results that LLVM cannot confirm in the other list. That is, the load operations that are *MustAlias* or *PartialAlias* with the store operation are stored in one list, while the load operations that are *MayAlias* with the store operation are stored in another list. We present the detailed algorithm, **Algorithm 1**, in the Appendix.

4.4 LLVM’s Reachability Analysis

Although the algorithm is sound, it introduces false positives due to the context-insensitive and flow-insensitive nature of the alias analysis. The former limitation can be addressed by inlining functions and the later limitation can be addressed by reachability analysis, which tells whether a load operation is reachable from a store operation. We discard the alias analysis results if a load is not reachable from a store.

Soundness Discussion. Reachability analysis does not affect the soundness of the analysis. In contrast, it reduces false positives by making the alias analysis flow-sensitive.

Application Scenario. We present a simple example where reachability analysis is applicable in **Listing 1**. The first element of a public array, `public_array`, affects the control flow twice at line 2 and line 4. However, the first branch is benign, as the array does not hold a secret at that time. Without reachability analysis, both branches are reported as CT violations as they read from addresses that are aliased with the store operation at line 3. With reachability analysis, we can determine that the first branch is not reachable from the store operation, so that we can filter the false positive.

```

1 func ( secret , public_array ) {
2   if ( public_array [0] ) { ; } Benign
3   public_array [0] = secret ;
4   if ( public_array [0] ) { ; } CT Violation
5 }
  
```

Listing 1: Benefit of reachability analysis.

LLVM’s Reachability Analysis. In principle, reachability analysis can be done by traversing the control flow graph

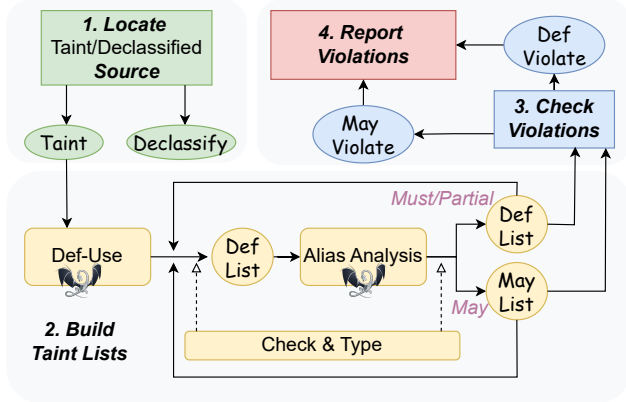


Figure 3: CT-LLVM’s CT Analysis Logic Overview.

(CFG) and checking whether one instruction is reachable from another. LLVM eases this process by providing an API that already implements the reachability analysis, returning true if the destination instruction is reachable from the source instruction, and false otherwise. Therefore, we only need to call the API with the two instructions and incorporate the results into the alias analysis to make it flow-sensitive.

5 CT-LLVM

In this section, we explain how we leverage the aforementioned LLVM features to implement CT-LLVM. The tool first transforms the IR code with `mem2reg` and function inlining to a form that is suitable for analysis. Then, it takes four stages to detect CT violations. We first explain the four-stage analysis logic. Then, we explain how we define the type system to reduce false positives. In the end, we explain how CT-LLVM supports proving the absence of CT violations and detecting CT violations in two different analysis modes.

5.1 CT Analysis Mechanism

We present a high-level diagram of the CT analysis logic in Figure 3. First, CT-LLVM locates the taint and declassify sources. Second, it builds two taint lists by leveraging the def-use chain and alias analysis. Third, it iterates each tainted instruction and checks whether it violates the CT policy. Finally, it generates a summary for each taint source and reports CT violations with source code information.

Stage 1 Locate Source. In the first stage, CT-LLVM locates the taint and declassify sources in two different ways. Taint source stands for variable that is marked as secret, while declassify source stands for variable that is marked as public. First, the user can specify which variable to taint or declassify by providing its name and type. This method is applicable to scenarios where the user has knowledge on what variables, including both function arguments and internal variables, are se-

cret or public. Alternatively, CT-LLVM assumes all function arguments are secret and taints all of them. This method is applicable to scenarios where the user cannot identify all secrets in a cryptographic library. It may introduce false positives due to over-approximation. However, CT-LLVM provides an analysis summary for each analyzed taint source, allowing the user to inspect the results of interest.

Stage 2 Build Taint Lists. After locating the taint and declassify sources, the analysis proceeds to track the explicit data flow of the taint source by building two taint lists. Both taint lists contain results of traversing the def-use chain, while one contains the MustAlias and PartialAlias results from the Alias Analysis and the other contains the MayAlias results.

CT-LLVM first recursively traverses the def-use chain of the taint source to build a list called the *Def List*. This list contains instructions that use values propagated from the taint source. Before inserting an instruction into the list, CT-LLVM checks if it is an immediate use of a declassified source and skips it if so. Each instruction in the *Def List* is typed to reduce false positives. In the next section, we elaborate on the type system. Since the SSA form is already flow-sensitive, CT-LLVM does not need to check the reachability of the inserted instruction in this step.

To track the propagation through memory operations, CT-LLVM enriches the *Def List* by leveraging the Alias Analysis. We have already illustrated the algorithm for doing so in Section 4.3. In practice, only store operations that store a non-declassified tainted variable into the memory are evaluated. Similarly, only those load operations that reachable from the store operation being evaluated are examined. In addition, we also leverage the LLVM intrinsic functions to track the taint propagation through `memcpy` and `memcpy` functions that has data length known at compile time.

LLVM’s alias analysis provides conservative results by returning *MayAlias*, which is the main source of false positives. To distinguish precise results from conservative results, we maintain two separate lists to keep the results of *MustAlias* and *PartialAlias* separate from *MayAlias*. The results of *MustAlias* and *PartialAlias* are merged with the *Def List* from the first step, while the results of *MayAlias* are kept in the other list, called the *May List*. Same as the first step, each instruction is checked for immediate use of declassified sources and typed when updating the corresponding list. We recursively traverse the def-use chain and query the Alias Analysis to complete the propagation of variables loaded from aliased addresses. We note that the def-use chain of tainted variables from the *May List* is also stored in the *May List*.

Stage 3 Check CT Violations. After obtaining two complete lists of tainted instructions, CT-LLVM then checks whether they breach the CT policy. By default, CT-LLVM checks for three types of CT violations. First, it checks whether a tainted variable, labelled as H , is used to compute the condition of a branch instruction, which can be leaked through branch predictor attacks [20, 46]. Second, it checks whether the address

of a store/load operation has been tainted and labelled as H . If so, the secret can be leaked through cache attacks [31, 44]. Third, it checks whether a tainted variable, which is labelled as H , is used as an operand of variable-time instructions, such as divisions. If so, the secret can be potentially leaked through timing attacks [4, 11]. In addition, the user could also specify other sources of CT violations, such as the `select` instruction, to fit the analysis for different threat models.

Stage 4 Report CT Violations. As the last stage of the CT analysis, CT-LLVM reports the found violations. First, the tool reports an analysis summary for each taint source, including their name in the source code and the number of each type of CT violations. Second, the tool locates the source code that violates the CT policy and prints the line number and the source code. In addition, the user could choose to dump the entire analysis procedure to reason how the tainted source propagates to the printed code. The dump reports each tainted source code and the reason of tainting it.

5.2 Type System

We now elaborate how we define the type system to reduce false positives. In a nutshell, we leverage a common assumption in the CT threat model that an address, which is not computed with secret, is public, while the value loaded from it is secret. Therefore, the goal of the type system is to distinguish whether a tainted variable is propagated from a public address or a secret value.

Motivating Example. We present a motivating example in Figure 4. On the left is the source code and on the right is the corresponding LLVM IR code. The function takes a pointer that points to a memory region where secrets are stored in. To prevent null pointer dereference, it first checks if the pointer is valid before reading the first element pointed by the pointer. If the pointer is valid, the function then checks if the first element pointed by the pointer is zero. Validating a pointer before dereferencing it is a common practice to avoid null pointer dereference and thus frequently appears in the source code. Apparently, validating a pointer does not leak secret information. However, our empirical study finds that both Ctchecker [47] and Flowtracker [33] report CT violations at line 2 and line 7. The former violation is a false positive while the latter one is a true positive. We note that Ctchecker does not use type system at all while Flowtracker uses a variation of flow-sensitive type system [23].

Typing Rules. Our type system has a semi-lattice structure, which has three types $\{H, L, \perp\}$. We use H for secret-related variables, L for public-related variables. By default, all variables are typed as \perp , indicating no knowledge of their type.

With this semi-lattice type system, we define three typing rules to systematically distinguish between public addresses and secret values. ❶ First, we consider addresses, including those pointed by pointers, (e.g., pointer, struct, array) are public, unless they are computed with secrets. ❷ Second, we type

1 <code>func(int* s_ptr){</code>	1 <code>define @func(i32* %0) {</code>
2 <code> if (!s_ptr)</code>	2 <code> L %1 = icmp eq i32* %0, null</code>
3 <code> return ; }</code>	3 <code> ⊥ br %1, EXIT, NEXT1</code>
4	4 <code> L NEXT1: %2 = getelementptr</code>
5	5 <code> i32* %0, i64 0</code>
6	6 <code> H %3 = load i32, i32* %2</code>
7 <code> if (s_ptr[0])</code>	7 <code> H %4 = icmp ne i32 %6, 0</code>
8 <code> return; }</code>	8 <code> ⊥ br i1 %4, NEXT, label %9</code>
9	9 <code> ⊥ NEXT2: br EXIT</code>
10	10 <code> ⊥ EXIT: RET</code>
11 <code>}</code>	11 <code>}</code>

C Code

LLVM IR

Figure 4: A motivating example for removing false positives with type system.

all variables loaded from the memory as H unless it has been previously declassified or labelled as L . ❸ Third, we type a variable as H if it is propagated from any variable that is labelled as H . The third typing rule is common in information-flow analysis, while the first two rules are explicitly defined for the CT threat model.

Application Demo. We explain how the type system removes the false positive while keeping the true positive with the LLVM IR code in Figure 4. CT-LLVM first taints the SSA value of `s_ptr` as the taint source. Since the taint source is never stored in the memory, simply traversing the *def-use chain* is sufficient to capture the entire data flow of the taint source. The taint source, `%0` has two immediate uses, one computes the branch condition at line 2 and the other computes the memory address at line 4. Since both of them are not loaded from the memory, CT-LLVM types them as L . According to our rules of detecting CT violation, the branch at line 2 is considered as benign. At line 6, the computed address is used to load a value from the memory. According to our second typing rule, a value loaded from memory is considered as secret. Therefore, CT-LLVM types the loaded value as H . It then propagates the typed value according to the third typing rule. Consequently, the branch condition at line 7 is inherently typed as H and is considered as a CT violation.

Customization. We implement the type system as a plugin of CT-LLVM. Therefore, it can be updated, replaced or even removed to serve for different threat models while not affecting the other parts of the analysis. For example, the user can remove the first typing rule if the threat model assumes addresses are secret.

5.3 Proof Mode

CT-LLVM provides two modes for CT analysis, *Proof Mode* and *Violation Finding Mode*, focusing on two analysis tasks. Although both modes can find CT violations, the *Proof Mode* further ensures the absence of CT violations if no violations

are found. In this section, we explain how CT-LLVM operates under the *Proof Mode*. We first define cases that are provable by CT-LLVM, then we state the features of CT-LLVM that are necessary for the sound analysis. In the end, we discuss the limitations of the *Proof Mode*.

Provable Cases. As a common limitation of modular static analysis, CT-LLVM cannot analyze functions that are defined in files that are not passed to the compiler. Therefore, we force inlining all functions within the analyzed function to ensure the soundness of the analysis. We say a function is provable by CT-LLVM if it does not contain following cases: ❶ recursive functions, including mutual recursion; ❷ functions that contain indirect calls; ❸ functions that contain inline assembly; ❹ functions that are declared but not defined; ❺ and memcopy that has run-time determined data length.

Necessary CT-LLVM Features. Our sound analysis is built on the ability to capture all explicit data flow of a taint source. Therefore, CT-LLVM enforces using the def-use chain and all results of alias analysis to track the propagation of the taint source under the *Proof Mode*. The user could specify the taint source or chose to taint all function arguments.

Limitations. The *Proof Mode* has two limitations. First, it cannot analyze all functions due to the aforementioned limitations. Second, CT-LLVM relies on the correctness and precision of the alias analysis implemented in LLVM.

5.4 Violation Finding Mode

The goal of the *Violation Finding Mode* is to cover cases that are not provable by the *Proof Mode*. Since it does not aim for a sound analysis, the user can selectively enable the alias analysis and function inlining to find CT violations while ignoring the impact of function calls and alias analysis. In the next section, we show that this mode is effective in finding CT violations just as other CT analysis tools.

6 Reliability Evaluation

Our focus in this paper is not confirming known CT vulnerabilities but automatically detecting CT violations and prove their absence in a large-scale manner. However, confirming known CT vulnerabilities is an effective way to demonstrate the tool is reliable. Therefore, following the routine in the literature, we first evaluate the reliability of CT-LLVM by confirming known CT vulnerabilities.

6.1 Benchmark Suite and Tool Selection

We build the benchmark suite by referring to previous works [7, 12, 13, 15, 16, 17, 25, 38, 39]. We find that most of the previous work target at two types of CT violations. That is, the table lookup operations in symmetric ciphers (e.g., AES) and branches in modular exponentiation for asymmetric ciphers (e.g., RSA, ECC). We thus pick five different algorithms

that have been practically exploited according to a survey of micro-architectural timing attacks in cryptography [29]. The vulnerable algorithms and their corresponding cryptographic scheme are presented in Table 1. Our interest is not in confirming the same algorithm that is implemented in different libraries. Therefore, we only pick one implementation for each algorithm in our benchmark suite. Besides the five algorithms, we also include a recently discovered vulnerability in Kyber [11], which leaks secrets through a secret-dependent division. In the end, our benchmark suite contains six different algorithms² that cover all three types of CT violations.

Tool Selection We select ctgrind [26] and Flowtracker [33] for comparison. We choose ctgrind as it is the only popular tool used by cryptographic library developers [24]. We choose Flowtracker as it is another LLVM-based tool that has a sound analysis mechanism. Another important reason is that we are able to install, configure and use it without any issues.

6.2 Evaluation Results

We run CT-LLVM in the *Violation Finding Mode*, without inlining functions, and report the results with different combination of LLVM features in Table 1. We mark the leakage source that has been exploited by practical attacks, and the number of found CT violations for each tool in the table.

Confirm Leakages. We confirm that all three tools can find CT violations that have been previously exploited. The exception is Flowtracker and the release version of valgrind do not support detecting variable-time CT violations. Interestingly, we find that CT-LLVM can detect all exploitable CT violations just by traversing the *def-use* chain. We note that it only takes less than 50 LoC to implement this analysis mechanism.

False Positive. We then manually inspect the false positives in the results, except for flowtracker which reports too many CT violations. We confirm that ct-grind does not report any false positive, which is expected as it only reports violations at lines that are actually executed. Furthermore, CT-LLVM only reports false positives in the case of ECDSA with sliding-window. We manually confirm that these false positives are caused by the conservative alias analysis in LLVM.

Variant Number of Found Violations. In addition to the false positives, our manual inspection also finds that ct-grind does not report all CT violations at lines of code that got executed. A simple example is presented in Listing 2, where x is marked as a secret and b is a secret-dependent variable. We find that ctgrind reports a violation at line 2 but not at line 3. This explains why ctgrind reports fewer violations than CT-LLVM in some tests

²The implementation we chose are from bearssl-0.6, gnupg-1.4.13, gnupg-1.4.14, openssl-1.0.1e, openssl-1.0.1e and PQClean-round3 in the order of the table.

Table 1: Results of Reliability Evaluation. We manually verify whether each violation is true or false positive for ctgrind and CT-LLVM. The number of them is reported in the form of *T/F*.

Cryptography	Implementation	Leakage Source	Flowtracker	ctgrind	CT-LLVM		
					Def	Def + MustAlias	Def + Alias
AES	T-table [10, 31]	Cache x 32	32	32/0	32/0	32/0	32/0
RSA	Square-and-Multiply [44]	Branch x 1	86	3/0	5/0	5/0	5/0
	Square-and-Multiply-Always [46]	Branch x 1	88	3/0	6/0	6/0	6/0
ECDSA	Montgomery-Ladder [5]	Branch x 1	94	2/0	2/0	2/0	2/0
	Sliding-Window[9]	Branch x 1	207	12/0	7/0	7/0	56/42
Kyber512	Division [11]	Variable-Time x 1	–	–	1/0	1/0	1/0

```

1 mark_secret(&x, sizeof(x));
2 b = table[x]; Detected
3 while (b) {...}; Not Detected

```

Listing 2: ctgrind only reports the first CT violation.

6.3 Summary of Reliability Evaluation

We note that this is expected that all tested tools perform well in finding known CT violations. Furthermore, the difference in the number of found violations is also expected, as each tool has its own analysis mechanism. Hence, finding more CT violations does not necessarily mean the tool is better than others. For example, although ct-grind finds fewer violations than CT-LLVM in testing RSA, it still rejects the leaky implementation. In fact, our manual efforts highlight the difficulty of fairly comparing different CT analysis tools by only counting the number of found violations.

7 Proving Kyber Constant-Time

In this section, We demonstrate how CT-LLVM works in the *Proof Mode* with an example of Kyber implemented in PQClean³. We choose Kyber, now known as ml-kem, as our example because it is selected by NIST as a standard post-quantum KEM algorithm. Therefore, it is crucial to ensure that Kyber is CT. We first detail the steps we take to evaluate Kyber, which also serves as a tutorial of using CT-LLVM. Then, we report the results of the proof, showing that Kyber’s rejection sampling violates the traditional CT policy.

7.1 Proof Procedure

To demonstrate the automation of CT-LLVM, we do not make efforts on annotating secrets in the code. Therefore, we configure CT-LLVM to taint all function arguments. CT-LLVM runs in the *Proof Mode* and inline all function calls.

We compile Kyber512 with *clang-18* under the optimization level *-O3* provided by the default Makefile. In addition,

³<https://github.com/PQClean/PQClean>

we pass the compilation flag *-fembed-bitcode* to embed LLVM IR into the object file. After obtaining the object file, we extract the LLVM IR code with the *llvm-dis* tool. The output is stored in several IR files, which corresponds to their own source files. We thus merge them into one IR file with the *llvm-link* tool to facilitate function inlining. Finally, we leverage the *opt* tool to invoke the CT-LLVM pass to analyze the merged IR file. We note that the entire analysis process does not require any source code modification or changes to the building process of the project. Furthermore, the tools we rely on to extract, merge and analyze the IR code is provided by the official LLVM toolchain.

7.2 Proof Results

We conduct our experiment on *i7-1165G7* CPU that runs Ubuntu 22.04TLS. The entire analysis procedure is automatic without any human efforts. In 5.5 seconds, CT-LLVM analyzes 45,316 lines of IR code and reports that 30 out of 41 functions are CT, no matter their inputs are secret or not. Since CT-LLVM reports each line of source code that reported as a CT violation, we can manually go through the results and confirm whether they are true or false positives.

SELECT Instruction. First, we find *select* instruction in four functions, *poly_tobytes*, *polyvec_tobytes*, *polyvec_compress* and *poly_compress*. We provide an example in Listing 3. The C code at line 2 checks whether *t0* is negative and adds *KYBER_Q* to *t0* if it is. While the source code uses bitwise operations, its LLVM IR code uses the *select* operation to implement the conditional addition. Hence, if the operation is lowered to a conditional branch, the condition is leaked through the branch. With our best efforts, we find that the *select* operation is always lowered to bitwise operations in this case. However, we cannot guarantee that this will always be the case, as compiler optimization strategy is complex and changes over time.

Non-Secret CT Violations. Second, we find function *verify* is considered as non-CT by CT-LLVM due to the over-approximation of taint sources. The function compares whether two arrays are equal by iterating the arrays in a loop, which leaks the length of the arrays. If the length of the ar-

```

1 poly_tobytes :
2   t0 += ((int16_t)t0 >> 15) & KYBER_Q;
3
4   %8 = icmp slt i16 t0, 0,
5   %9 = select i1 %8, i16 KYBER_Q, i16 0,
6   %10 = add i16 %9, %7,

```

Listing 3: select instruction for & operation.

rays is a secret, then it leaks through the number of iterations. Otherwise, the function is CT.

Rejection Sampling. Third, we find five functions are rejected by CT-LLVM as they involve rejection sampling when generating random numbers. Rejection sampling checks whether the generated number is within a range and regenerates a new number if it is not. Hence, conditional branches are used to check whether a random number is within a range. Although these branches do violates the traditional CT policy, it is still CT under a probabilistic notion of CT [3].

False Positive. In the end, we find that CT-LLVM reports one false positive in the function *shake256_rkprf* due to the conservative alias analysis in LLVM. LLVM’s alias analysis considers two array indexes may aliased, although they are not at run time. A detailed discussion is in Appendix A.

Summary. To summarize, CT-LLVM automatically proves 30 out of 41 functions are always CT without annotating secrets in the code. Four functions are rejected because they contain select instructions and one function is rejected due to the over-approximation of taint sources. One function is rejected due to the conservative alias analysis in LLVM. The rest five functions are not CT due to the use of rejection sampling. Giving the fact that only one false positive is reported among 41 tests, we believe that CT-LLVM is reliable in proving CT in cryptographic libraries. We provide a full list of the proved functions in Table 4.

8 Large-Scale Analysis

In this section, we leverage the technique we present in the last section to automatically analyze nine popular cryptographic libraries.⁴ Due to the limitation of static information-flow analysis, CT-LLVM cannot soundly analyze all functions. Therefore, the first goal of our large-scale analysis is to understand the number of proveable functions in the wild and the main reasons that make functions to be unprovable. The second goal is to investigate how many functions can be automatically proved to be CT without human efforts of annotating secrets in the code. The third goal is to find CT violations in the libraries. Due to the page limit, we only selectively report CT violations that we manually verified.

⁴gmssl (commit:34fa519dc0f94a9a3995d9daf09c84cdac37abd8), bearssl-0.6, wolfssl-5.7.6, s2n-tls-1.5.10, libgcrypt-1.11.0, mbedtls-3.6.2, Tongtsuo-8.4.0, openssl-3.4.0, boringssl-0.20241024.0

8.1 Analysis Setup

Same as the setup in the Kyber example, we configure CT-LLVM to run in the Proof Mode and taint all function arguments as secrets. In addition, we set a limitation on the number of inlining levels and the number of memory operations for analysis to stop the analysis when it takes too long to finish. Specifically, we set the maximum inlining level to 10 and the maximum number of memory operations for analysis to 2,000. Therefore, the functions we prove are not only fully inlined but also under a certain size.

8.2 Analysis Results

We present the statistics of the analysis results in Table 2. In the table, we list the number of functions in each library (*All*), and the percentage of provable functions (*Provable*). We compute the percentage of functions verified by CT-LLVM (*CT*) with respect to all functions and provable functions, respectively. In addition, we report the time cost for analyzing each library in seconds.

Table 2: Statistics of Automatic Proof Analysis on Cryptographic Libraries.

Library	All	Provable	CT		Time (seconds)
			All	Provable	
GmSSL	1550	33%	9%	28%	87
BearSSL	1268	70%	48%	69%	145
wolfSSL	1802	42%	17%	41%	38
s2n-tls	1911	17%	14%	81%	14
Libgcrypt	2716	35%	22%	62%	52
MbedTLS	1548	33%	15%	45%	23
BoringSSL	5624	29%	21%	72%	37
Tongtsuo	12576	32%	25%	77%	220
OpenSSL	14351	32%	24%	77%	290

Percentage of Provable Functions. We find that the percentage of provable functions is not consistent across different libraries. In contrast, it varies from 17% to 70%, with an average of 36% of the functions are provable. In fact, such a large variance is expected, as libraries, such as *s2n-tls*, that relies on external cryptographic libraries, have fewer functions that can be inlined. On the other hand, having 36% of the functions provable is a good start towards automatically proving CT for all functions. Specifically, the ability of proving 70% of the functions in *BearSSL* is a very promising result. This shows that cryptographic libraries can be designed and implemented in a way that facilitates large-scale automatic CT analysis.

Percentage of CT Functions. Compared to the various percentage of provable functions, the percentage of CT functions is more consistent across different libraries. On average, 61% of the provable functions are CT. Recall that we over-approximate the taint sources to facilitate the automatic analysis, which introduces false positives. Therefore, our results

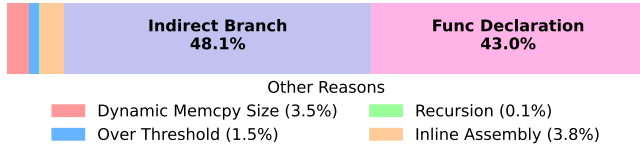


Figure 5: Distribution of reasons for unprovable functions.

show that mainstream cryptographic libraries do make efforts to implement their functions in a CT manner.

Analysis Time. The time cost of analyzing libraries increases with the number of analyzed functions. However, we note that the analysis overhead is relatively low. Specifically, it takes less than three minutes to analyze 70% of the functions in BearSSL. The low overhead of the analysis process suggests that CT-LLVM can be integrated into the development pipeline with a relatively low cost.

8.3 Reasons for Unprovable Functions

We then investigate the reasons that make functions unprovable and present the distribution of the reasons in Figure 5. As shown in the figure, two main reasons make functions unprovable are the use of indirect branches and external functions that cannot be inlined. They account for more than 90% of the unprovable functions. This also indicates that a cryptographic library using fewer indirect branches and external functions can have more functions to be soundly analyzed. According to Table 2, BearSSL is such a library that implements fewer indirect branches and external functions, which makes 70% of its functions provable. Besides, the use of inline assembly and using run-time determined memcpy size also make functions unprovable. We note that memcpy whose size is determined at runtime cannot be soundly analyzed by static analysis, unless we assume a maximum size for the memcpy operation. In the end, only 1.5% of the unprovable functions are due to the limitation we set to prevent the analysis from taking too long to finish. Compared to the other reasons, this number is small enough to be ignored.

8.4 Report CT Violations

Based on the analysis summary, we find two interesting facts about the CT violations in the libraries. First, the libraries adopt fewer CT implementations to functions that are not directly used in cryptographic schemes. Second, well-known CT vulnerabilities still exist in some libraries. We now report several CT violations that we manually verified.

wolfSSL. We manually inspect the results of analyzing `misc.c` in `wolfssl-5.7.6`. We find several non-CT arithmetic (e.g., comparison, addition) functions and data-type conversion functions. An example of vulnerable comparison function is shown in Listing 4, which leaks the minimum value of two inputs under 00 flag.

```

1 word32 min(word32 a, word32 b) {
2     return a > b ? b : a;
3 }

```

Listing 4: Non-CT Comparison.

Another example is the non-CT ByteToHex conversion as shown in Listing 5. Although the array takes less than a cache line size, partial bits of the input can still be leaked through sub-cache-line accesses [35, 36].

```

1 char ByteToHex(byte in) {
2     char kHexChar[] = {'0', '1', '2', '3', '4', '5', '6',
3     '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
4     return (char)(kHexChar[in & 0xF]);
5 }

```

Listing 5: Non-CT Byte to Hex Conversion.

S2n-tls. We find the library using a vulnerable version of reading and writing base64 encoded data, implemented in file `s2n_stuffer_base64.c`. Similar functions in other libraries have been exploited by previous works [35, 36]. The code in Listing 6 checks if the input is a base64 character.

```

1 static const uint8_t b64_inverse[256] = {...};
2 bool s2n_is_base64_char(unsigned char c) {
3     return (b64_inverse [*((uint8_t *) (&c))]
4             != 255);
5 }

```

Listing 6: Non-CT Check Base64 Format.

GmSSL. First, the library contains vulnerable base64 encoding and decoding functions (link) that are vulnerable to sub-cache-line attacks [35, 36]. Second, it uses secret-dependent division for the poly-compression function in the Kyber implementation (link). Third, although the library leverages constant-time hardware instructions (such as AES-NI and AVX512) to implement SM4, it still uses an S-box lookup table for the key scheduling and encryption (link). In the end, we find a few non-CT implementations, such as non-CT comparison (link) for SM2 signature scheme in the library. Although some of the non-CT implementations, has its CT version in the assembly code, they are not used by default.

9 Detect Compiler-Introduced CT Violations

In the end, we show that CT-LLVM helps detect compiler-introduced CT violations. The methodology relies on the observation that currently identified LLVM-related compiler-introduced CT violations are due to the lowering of the `select` instruction to conditional branches. Hence, as we have illustrated in Section 3.2, capturing the `select` instruction in the LLVM IR is a cheap and effective way to detect such violations. In this section, we conducted a large-scale study and report the status of using `select` in the wild. Then we report violations that are detected by CT-LLVM.

9.1 Select Instruction In the Wild

We configure CT-LLVM in the Proof Mode and only analyze functions that can be soundly analyzed. We trigger LLVM issues `select` instructions, by compiling libraries with `O3` optimization level. We report the results in Table 3, where we present the number of analyzed functions and the percentage of functions that contain `select` instructions. According to the table, we can see that the `select` instruction is relatively common in most of the tested libraries.

In fact, the idea of selecting a value based on a condition is a common mitigation to defeat timing attacks. Such mitigation is usually built by storing the condition in a mask and using branchless operations, such as `val |= a & mask`, to selectively update the value. Under higher optimization levels, LLVM may detect such sophisticated operations can be simplified into a `select` instruction. In some cases, LLVM lowers the `select` instruction into conditional moves, which are CT. However, on platform that do not support conditional moves, or compiler believes branches are better than conditional moves, the `select` instruction is lowered to branches. It is hard to anticipate whether and when the `select` instruction is lowered to a branch. Therefore, a conservative but safe approach is to prevent the `select` instruction from appearing in the code.

Table 3: Number of Functions that contains `select`

	GmSSL	BearSSL	wolfSSL	s2n-tls	Libgcrypt
#Funcs	430	864	752	164	873
#Select	9%	5%	19%	0.03%	9%
	MbedTLS	BoringSSL	Tongsuo	Openssl	
#Funcs	498	786	2602	2907	
#Select	19%	10%	7%	8%	

9.2 Report CT Violations

We manually analyze partial results and confirm that `select` instructions tend to appear when fixing a CT violation or trying to make the code CT with the mask technique. In this section, we report three interesting cases that we have verified the `select` instruction can be lower to conditional branches with magic flag `x86-cmov-converter-force-all` in `clang18`.

New Violation after Fixing One. First, we find that fixing a CT violation can sometimes lead to a new violation. For example, `wolfSSL` fixes the CT vulnerability in the Base64 conversion by always accessing two cache lines and using a mask to select the value from two cache lines ([link](#)). We find that under `O3` optimization, a `select` instruction is used to conditionally select the value from two cache lines.

(Non-)CT Function Second, we find that functions particularly designed to be CT are not CT under `O3` optimization. For example, function `sp_clamp_ct` in `wolfSSL`, which is

frequently used in big number arithmetics, is particularly designed to be CT ([link](#)). Similarly, the use of the mask is also converted into a `select` instruction by LLVM.

Another interesting case we find is the finite-field multiplication for AES in `GmSSL`. As shown in the code snippet in [Listing 7](#), when the conditional move is lowered to a branch, it leaks the top bit of the input.

```
1 static uint8_t x2(uint8_t a) {
2     return (a >> 7) ?
3         ((a << 1) ^ 0x1b) : (a << 1);
4 }
```

Listing 7: Finite Field Multiplication for AES.

9.3 Discussion

Exhausting all compiler-introduced CT violations is challenging as it is affected by many factors. For example, we find a `select` operation in `BoringSSL` where converting ASCII to binary is explicitly written in a CT manner ([example](#)). However, such operation only exist with `llvm14` instead of `llvm18`. We confirm that the operation is lowered to branches under flag `-O3 -m32 -march=i386` in `clang14`. Since CT-LLVM works with LLVM14 to the latest version at the time of writing (LLVM19), it is possible to try different versions of clang and different optimization levels to detect more CT violations.

10 Conclusion

In this paper, we take a significant step toward automatically analyzing cryptographic libraries for constant-time properties. We propose and implement a new CT analysis tool, CT-LLVM, that is usable, maintainable and scalable for automatic and sound CT analysis. With the large-scale study, we show that on average 36% of the functions in cryptographic libraries can be soundly analyzed without human efforts, and 61% of the analyzed functions are CT. Based on the large-scale analysis results, we also find new CT violations in the source code or due to compiler optimizations.

According to the findings reported in this paper, several future works can be done with CT-LLVM. First, it is possible to annotate cryptographic code, and reason about inline assemblies to get more functions soundly analyzed. Second, it is possible to extend the analysis of CT properties to speculative CT properties, by updating the violation detection rules of CT-LLVM. Third, since CT-LLVM is an LLVM plugin and works with multiple versions of LLVM, more comprehensive analysis can be done to understand which version of LLVM under which optimization level can introduce CT violations. Finally, an interesting direction is to transparently lift binary code into LLVM IR, so that CT-LLVM can analyze the CT properties of binary code.

11 Responsible Disclosure

We have shared our findings and our tool ⁵ with the nine cryptographic libraries that we have analyzed in this paper. We first report the results of the responsible disclosure for libraries that we have identified CT violations in this paper. Then we report the status of the responsible disclosure for the rest of the libraries. During the discussion with wolfSSL and GnuPG teams, we found and reported two new CT violations in wolfSSL and Libgcrypt. This shows that completely avoiding CT violations in a large codebase is challenging.

wolfSSL. WolfSSL acknowledged our findings and fixed the reported source-code level CT violations in a recent pull request (PR). Currently, wolfSSL is working on fixing compiler-introduced CT violations reported in this paper.

During the discussion, we helped check whether a non-table-based implementation of Base64 decoding is CT (link). Interestingly, CT-LLVM reports that the implementation is CT at the source-code level but may not be CT with compiler optimizations, due to the use of `select` instruction. Our further investigation confirms that under `O3` and `x86-cmov-converter-force-all` flags, branches are introduced to the binary code. This again highlights the difficulty of writing CT code. A demo is available at [here](#).

Additionally, we would like to thank the wolfSSL team for trying CT-LLVM when fixing the CT violations and sharing their experience with us.

s2n-tls. AWS Security quickly responded to our report and fixed the reported CT violation in a recent pull request (PR).

BoringSSL. We have reported the found compiler-introduced CT violations in BoringSSL through the Google Bug Hunter system. The responsible team has labelled the reported issue with severity 2 and is considering the fix.

GmSSL. We have contacted the maintainer of GmSSL on 23/01/2025 and reported the found CT violations. Till the time of publication, we have not received any response.

Libgcrypt. We have contacted the security team of Libgcrypt on 23/01/2025 and shared our research findings. The security team thanked us for sharing the findings. During the follow-up discussion, we have notified the security team that `_gcry_mpih_cmp_ui` implemented in file `mpih_cmp_ui` is not CT. The security team acknowledged the issue and has fixed it in a recent commit (link).

MbedTLS, OpenSSL. MbedTLS thanks us for sharing the research findings and OpenSSL has shared our findings to the `openssl-security` list for consideration.

BearSSL, Tong suo. We have contacted the maintainers of BearSSL and Tong suo on 23/01/2025 and shared the manuscript of this paper. Till the time of publication, we have not received any response.

⁵Available at <https://github.com/Neo-Outis/CT-LLVM-Artifact>

References

- [1] Onur Aciğmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *IACR Cryptol. ePrint Arch.*, page 351, 2006.
- [2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, pages 53–70, 2016.
- [3] José Bacelar Almeida, Denis Firsov, Tiago Oliveira, and Dominique Unruh. Leakage-free probabilistic jasmin programs. In *CPP*, pages 3–16, 2025.
- [4] Marc Andryscio, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE SP*, pages 623–639, 2015.
- [5] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ECDSA with less than one bit of nonce leakage. In *CCS*, pages 225–242, 2020.
- [6] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. Sidetrail: Verifying time-balancing of cryptosystems. In *VSTTE*, volume 11294, pages 215–228, 2018.
- [7] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. Abacus: Precise side-channel analysis. In *ICSE*, pages 797–809, 2021.
- [8] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *IEEE SP*, pages 777–795, 2021.
- [9] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES*, volume 8731, pages 75–92, 2014.
- [10] Daniel J Bernstein. Cache-timing attacks on aes. 2005.
- [11] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. Kyber-slash: Exploiting secret-dependent division timings in kyber implementations. *IACR Cryptol. ePrint Arch.*, page 1049, 2024.
- [12] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. In *ESORICS*, volume 10492, pages 260–277, 2017.
- [13] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *IEEE SP*, pages 505–521, 2019.

- [14] Luwei Cai, Fu Song, and Taolue Chen. Towards efficient verification of constant-time cryptographic implementations. In *FSE*, pages 1019–1042, 2024.
- [15] Sudipta Chattopadhyay and Abhik Roychoudhury. Symbolic verification of cache side-channel freedom. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11): 2812–2823, 2018.
- [16] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *IEEE SP*, pages 1021–1038, 2020.
- [17] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *PLDI*, pages 406–421. ACM, 2017.
- [18] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*, pages 431–446, 2013.
- [19] Vijay D’Silva, Mathias Payer, and Dawn Xiaodong Song. The correctness-security gap in compiler optimization. In *IEEE SPW*, pages 73–87, 2015.
- [20] Dmitry Evtushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, pages 40:1–40:13, 2016.
- [21] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "these results must be false": A usability evaluation of constant-time analysis tools. In *USENIX Security*, 2024.
- [22] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In *CCS*, pages 1690–1704, 2023.
- [23] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, pages 79–90, 2006.
- [24] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they’re not that hard to mitigate": What cryptographic library developers think about timing attacks. In *IEEE SP*, pages 632–649, 2022.
- [25] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. Cache refinement type for side-channel detection of cryptographic software. In *CCS*, pages 1583–1597, 2022.
- [26] Adam Langley. Ctgrind. <https://www.imperialviolet.org/2010/04/01/ctgrind.html>, 2010.
- [27] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015.
- [28] LLVM. Llvm implementation of alias analysis. Online, n.d. URL <https://llvm.org/docs/AliasAnalysis.html#available-aliasanalysis-implementations>.
- [29] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2022.
- [30] Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Sören van der Wall, and Zhiyuan Zhang. Transparent decompilation for timing side-channel analyses. *arXiv preprint*, 2025.
- [31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006.
- [32] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, pages 1697–1702, 2017.
- [33] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *CC*, pages 110–120, 2016.
- [34] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking bad: How compilers break constant-time~implementations. *CoRR*, abs/2410.13489, 2024.
- [35] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util::lookup: Exploiting key decoding in cryptographic libraries. In *CCS*, pages 2456–2473, 2021.
- [36] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. Teejam: Sub-cache-line leakages strike back. *CHES*, 2024:457–500, 2024.
- [37] Laurent Simon, David Chisnall, and Ross J. Anderson. What you get is what you C: controlling side effects in mainstream C compilers. In *EuroS&P*, pages 1–15, 2018.
- [38] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *USENIX Security*, pages 235–252, 2017.
- [39] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. In *USENIX Security*, pages 657–674, 2019.
- [40] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Security*, pages 603–620, 2018.
- [41] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding

```

1 keccak_inc_absorb( state , r, key, mlen):
2   while (mlen + state [25] >= r) {
3     for (i = 0; i < r - state [25]; ++i) {
4       state [( state [25] + i) >> 3] ^=
5         (uint64_t)key[i] << ...;
6     }
7     mlen -= (size_t)(r - state [25]);
8     m += r - state [25];
9     state [25] = 0;
10  }

```

Listing 8: False Positive Example

side channels in binaries. In *ACSAC*, pages 161–173, 2018.

- [42] Jan Wichelmann, Florian Sieck, Anna Pättschke, and Thomas Eisenbarth. Microwalk-ci: Practical side-channel analysis for javascript applications. In *CCS*, pages 2915–2929, 2022.
- [43] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *USENIX Security*, pages 3655–3672, 2023.
- [44] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.
- [45] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.
- [46] Zhiyuan Zhang, Mingtian Tao, Sioli O’Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. Bunnyhop: Exploiting the instruction prefetcher. In *USENIX Security*, pages 7321–7337, 2023.
- [47] Quan Zhou, Sixuan Dang, and Danfeng Zhang. CtChecker: A precise, sound and efficient static analysis for constant-time programming. In *ECOOP*, volume 313 of *LIPICs*, pages 46:1–46:26, 2024.

A False Positive in Proving Kyber CT

We present a simplified version of the code in Listing 8, where `state[25]` stores the number of bytes that have been absorbed but not permuted. Although its update does not depend on the secret data (`key`) at run time, LLVM’s alias analysis considers the address of `state` at line 4 may aliased with the address of `state[25]` at line 2 and line 3. Therefore, it considers the function as non-CT.

B Full List of Provable Kyber512 Functions

We present the proof results of all provable Kyber512 functions in Table 4. We mark a function is proved to be CT with ✓, not CT with ✗, and false positive with ●. In addition, we also report the number of evaluated IR instructions and the time cost in milliseconds for each function.

Table 4: Kyber512 Proof Results

Function	CT	Instructions	Time (ms)
montgomery_reduce	✓	7	0.136
barrett_reduce	✓	9	0.139
poly_reduce	✓	11	0.195
poly_tomont	✓	11	0.197
cmov_int16	✓	13	0.318
poly_tomsg	✓	15	0.288
poly_cbd_eta2	✓	17	0.364
polyvec_reduce	✓	23	0.315
poly_frombytes	✓	23	0.282
poly_cbd_eta1	✓	24	0.445
poly_frommsg	✓	32	0.382
ntt	✓	32	0.452
poly_compress	✗	39	0.374
polyvec_decompress	✓	41	0.391
polyvec_frombytes	✓	48	0.46
kyber_shake128_absorb	✓	49	1.355
poly_decompress	✓	52	0.387
poly_ntt	✓	53	0.532
poly_add	✓	53	0.496
poly_sub	✓	53	0.493
polyvec_compress	✗	55	1.196
poly_tobytes	✗	59	0.459
poly_invntt_tomont	✓	74	0.641
invntt	✓	74	0.651
verify	✗	98	0.701
polyvec_ntt	✓	107	1.021
polyvec_add	✓	109	0.816
polyvec_tobytes	✗	120	0.936
polyvec_invntt_tomont	✓	149	1.311
basemul	✓	269	1.691
poly_basemul_montgomery	✓	620	4.478
poly_getnoise_eta2	✓	846	13.614
indcpa_dec	✓	880	15.318
polyvec_basemul_acc_montgomery	✓	1282	12.394
poly_getnoise_eta1	✓	1640	33.499
gen_matrix	✗	1702	54.996
kyber_shake256_rkprf	●	2215	21.36
indcpa_keypair_derand	✗	3756	434.695
indcpa_enc	✗	7292	1208.916
crypto_kem_enc_derand	✗	9688	1493.274
crypto_kem_keypair_derand	✗	13676	2181.493

Algorithm 1 Alias Analysis for CT Analysis

```
1: taint_chain  $\leftarrow$  def-use-chain of secret
2: load_Instrs  $\leftarrow$  load instructions in target function
3: alias_list, mayalias_list  $\leftarrow$   $\emptyset$ 
4: for store_op  $\in$  taint_chain do
5:   stored_value  $\leftarrow$  value stored by store_op
6:   if stored_value  $\in$  taint_chain then
7:     for load_op  $\in$  load_Instrs do
8:       if load_op  $\notin$  taint_chain then
9:         store_ptr  $\leftarrow$  address of store_op
10:        load_ptr  $\leftarrow$  address of load_op
11:        alias  $\leftarrow$  AA: store_ptr and load_ptr
12:        if alias  $\in$  {MustAlias, PartialAlias} then
13:          insert load_op to alias_list
14:        end if
15:        if alias  $\in$  {MayAlias} then
16:          insert load_op to mayalias_list
17:        end if
18:      end if
19:    end for
20:  end if
21: end for
```
