A Quantitative Evaluation of Constant-Time Analysis Tools

Zhenzhi Lai*‡, Zhiyuan Zhang†, Tian Qiu*‡, Van-Thuan Pham*§ and Udaya Parampalli*§
*The University of Melbourne, Australia
†Max Planck Institute for Security and Privacy, Germany
‡{zhenzhil,tq}@student.unimelb.edu.au †zhiyuan.zhang@mpi-sp.org §{thuan.pham,udaya}@unimelb.edu.au

***Abstract*—The constant-time programming policy protects cryptographic code from timing side-channels. However, writing constant-time code remains challenging even for experienced developers. Therefore, over 50 CT analysis tools have been published to help them identify constant-time violations. While several studies have examined those tools, they predominantly rely on qualitative observations and case studies, rather than quantitative assessments. Therefore, we present *AutoBench-CT*, a modular benchmark that manually verified per-violation ground truths across 64 cryptographic tasks and the first unified quantitative metrics to quantify the *Operational Performance*, *Detection Quality*, and *Usability* for seven such tools. With the results, we quantitatively show why CT-grind, the first tool published in 2010, is still the most popular tool in the wild. Based on the quantitative findings, we further provide new insights into recurring limitations across tools and propose a composite pipeline to maximize the security guarantees of constant-timeness in practice.**

***Index Terms*—Constant-time analysis tools. Timing attacks.**

1. Introduction

Micro-architectural timing attacks exploit the secret-dependent execution time of a program, typically a cryptographic implementation, to infer sensitive information such as secret keys [1], [2], [3], [4]. For example, the two paths of a secret-dependent branch may execute different numbers of instructions, resulting in observable execution time variations based on the branch condition. To prevent such attacks, a widely adopted countermeasure is the *constant-time (CT) programming policy* [5]. It disallows three classes of secret-dependent behavior: memory accesses, conditional branches, and variable-latency operations (e.g., divisions). The CT policy has since become the *de facto* programming rule for modern cryptographic libraries.

To help cryptographic code developers write CT code, various approaches, such as dynamic analysis [6], [7], [8], symbolic analysis [9], statistical analysis [10], and static analysis [5], [11], [12], [13], have been proposed to either detect CT violations or prove their absence. To date, over 50 CT analysis tools have been published [14]. Despite the increasing number of published tools, a recent user study [15] points out that the most popular tool is still CT-grind [6], which was published in 2010. Its popularity can be partly explained by its strong usability, as CT-grind is integrated into Valgrind [16]; yet, it remains unclear whether the tool still provides state-of-the-art (SOTA) CT analysis performance. To further avoid the bias of any single tool, several works since 2025 [17], [18], [19] have combined multiple CT analyzers to strengthen security assurance. However, such compositions are often guided by intuitive assumptions about tool complementarity, rather than comprehensive quantitative evaluations. After all, numerous tools have been published in the past 16 years. Since each new tool has typically been evaluated only on different benchmarks with non-uniform evaluation settings at the time of publication, the relative strengths and limitations of these tools remain underexplored. Recent studies [15], [20], [21] have recognized this gap and called for a unified benchmark to systematically evaluate and compare CT analysis tools. Such a benchmark would ultimately help developers choose the most suitable tool for their needs.

However, evaluating CT analysis tools is challenging. An evaluation needs to be compatible with different tools and programs under test, and provide ground truth for precise judgment of each reported violation. While these two prerequisites are relatively well studied by the broader software testing and verification community [22], [23], [24], [25], a less studied yet critical requirement is a set of **quantitative metrics** that comprehensively reflect a tool's CT analysis performance. Although several works [26], [15], [20], [21] have evaluated CT analysis tools, the evaluations primarily rely on **qualitative discussion** rather than such metrics, leaving developers without a principled basis to choose between tools.

To address the gap, we present *AutoBench-CT*, a unified benchmark for quantitatively evaluating CT analysis tools. It adopts the modular design of widely used testing and verification benchmarks [22], [25], so tools and analysis targets can be plugged in through unified interfaces. A key challenge for quantitative evaluation is the lack of reusable per-violation ground truths and appropriate metrics. Thus, we spend significant initial effort in labeling ground truths and making them reusable in the future. Specifically, *AutoBench-CT* first applies cross-comparison across tools to automatically classify each reported violation as a true or false positive, and then allows users to manually correct them. To maximize the reliability of manual labels, we

† Corresponding author.

follow the practice in prior work [27] to cross-check the results between two participants with expertise. For metrics, we synthesize the evaluation criteria and concerns raised in prior benchmarks [26], [20], user studies [15], [21], and tool papers [7], [28], [8], [29], [30], [13], [31], [32] into a unified set of quantitative indicators that capture *Operational Cost*, *Detection Quality*, and *Usability*.

With *AutoBench-CT*, we evaluate six SOTA binary-level and one LLVM-IR-level CT analysis tools: CT-grind [6], Microwalk [28], DATA [8], Binsec [9], Pitchfork-angr [33], Abacus [29], and MemSan [34], across 64 analysis tasks. The analysis targets span a broad set of cryptographic schemes, including currently used symmetric and asymmetric primitives, as well as post-quantum algorithms designed for the near future. Based on the quantitative evaluation results, we aim to answer the following questions:

RQ1: Which tool achieves the strongest overall performance across all evaluation metrics?

RQ2: What are the most recurring limitations shared across different tools?

RQ3: How should CT analysis tools be combined to maximize security value in practice?

Surprisingly, we quantitatively show that CT-grind, the first CT analysis tool published in 2010 [15], outperforms the other evaluated tools on both operational cost and detection quality, with the highest precision, the highest relative recall, and a 100% self-termination rate within the resource budget. Its main weakness is report verbosity, but it is partly mitigated by a plaintext output option. For RQ2, we identify four recurring limitations: exhaustive path enumeration as the dominant resource cost, incomplete data-flow tracking, weak report interpretability with no runtime feedback, and a previously undocumented failure mode in which the analysis tool produces "mirage" violations (4% in CT-grind, 44% in MemSan). Those violations do not exist in the original program but can silently flip a program's CT/non-CT classification. For RQ3, our ablation shows that the taint+symbolic pipeline shared by three recent composite pipelines [18], [17], [19] is not the empirically optimal composition. Instead, taint+trace composition (the "CT-DATA" pipeline) reaches 0.974 recall on 94% tasks finished in-budget. Symbolic execution retains a unique role in verifying the absence of violations, but should be applied carefully only to small fragments after the dynamic pipeline reports no violations.

To summarize, we make the following contributions:

1) We proposed a set of evaluation metrics for quantitatively evaluating CT analysis tools and implemented an extensible benchmarking framework, *AutoBench-CT*. With it, we manually labeled 2227 ground-truth labels across 64 analysis tasks and evaluated six binary-level and one LLVM-IR-level SOTA tools.
2) We quantitatively demonstrate that CT-grind, the first CT analysis tool proposed in 2010, performs the best according to our evaluation metrics. We further identify recurring limitations and new insights across evaluated tools, providing actionable guidance for both tool developers and tool users.
3) Through a quantitative ablation study, we proposed a composite pipeline that maximizes the security guarantees of constant-timeness in practice.

The artifact is available on Link.

Listing 1 Finding all CT violations.

```
1 func(uint8_t sec, int pub[256]){
2     int index = pub[sec];
3     int index2 =pub[index & 0xff];
4     for(int i=0; i<256; i++){
5         if (sec == i)
6         pub[sec]++;
7     }
8 }
```

Tool	Line 2	Line 3	Line 5	Line 6
Abacus	✓		✓	✓
Binsec	✓		✓	
CT-grind	✓		✓	✓
DATA	✓	✓	✓	
Pitchfork	✓	✓	✓	
Microwalk	✓	✓	✓	

TABLE 1: We use ✓ if the violation is reported by the tool.

2. Motivating Example

We motivate the need for fine-grained, ground-truth-based evaluation with a small example. Listing 1 shows a non-CT C function. Let the input byte `sec` be the secret value. The function consists of four CT policy violations (highlighted): Line 2 and Line 6 directly access memory with the secret; Line 3 accesses memory with a secret-dependent value; Line 5 performs a control-flow transfer based on the secret directly. We compile the function for x86-64 with `Clang-18 -O0` (and additionally with `-m32` for Abacus only for compatibility) and run six SOTA tools: CT-grind[6], DATA[8], Microwalk[28], Binsec[9], Pitchfork-angr [33], and Abacus [29]. We mark `sec` as secret when necessary and do not mark any variable as public.

We present the analysis results in Table 1. Counterintuitively, although one might expect SOTA CT analysis tools to detect all violations in such a simple example, none do so completely. Prior benchmarks by Lopes et al. [26] and Geimer et al. [20] fail to expose this disagreement. Lopes et al. only evaluate whether a program is classified as CT or non-CT, under which all six tools appear equivalent. Geimer et al. compare only the number of reported violations, making CT-grind, Pitchfork-angr, Abacus, DATA, and Microwalk appear equally effective despite reporting different violation locations. Our results show that these disagreements become visible only through fine-grained, ground-truth-based evaluation at the per-violation level.

3. Background and Related Work

We introduce the CT policy, summarize the methodologies of existing CT analysis tools, review prior benchmarking efforts and findings on factors affecting tool performance, and discuss recent composite CT analysis pipelines.

3.1. Constant-Time Policy

The CT policy [5] is designed to prevent microarchitectural timing attacks [3], [4], [2] by ensuring that secret-dependent operations do not influence a program's execution time. Such attacks typically exploit variations in memory access patterns via data caches [1], [2], control-flow behavior via branch predictors [35], [36] and instruction caches [37], [38], and timing differences introduced by variable-time operations [39], [40], [41]. Therefore, the CT policy prohibits secrets from affecting memory accesses, control-flow transfers, and variable-time operations.

3.2. CT Analysis Techniques

Since 2010, dozens of techniques have been proposed for CT analysis. In general, CT analysis tools fall into two families. *Dynamic* analysis (taint analysis [6], [42], trace comparison analysis [43], [8], [44], [28], and statistical analysis [45], [10], [46], [7], [32]) concretely executes the program, which is generally scalable, but its completeness depends on input coverage. *Static* analysis (logical reduction [47], [5], [48], [49], [30], abstract interpretation [50], [51], [52], [53], [54], symbolic execution [55], [56], [57], [58], [9], [29], [59] and type-based information-flow analysis [11], [60], [12], [13]) reasons about all execution paths in principle, which can prove CT under various soundness guarantees, but typically at the cost of scalability. The six tools we evaluate (Section 5.2) span four of these subtechniques in both families. Complementary to CT analysis tools, CT compilation [61] takes a potentially non-CT program as input, along with the secret annotations, and outputs a CT binary. In this paper, we focus exclusively on CT analysis tools.

3.3. Related Work of Evaluating CT Analysis Tools

In contrast to the large number of available analysis tools, the work that evaluates them is limited. So far, existing tools have been evaluated by two unified benchmarks [26], [20], two user studies [20], [15], and work that proposes new analysis tools [7], [28], [8], [29], [30], [13], [31], [32]. We first introduce related work and then summarize the identified limitations of CT analysis tools.

Benchmarks. Lopes et al. [26] first built a unified benchmark that consists of over 60 cryptographic programs to evaluate CT analysis tools. It performs coarse-grained evaluation and focuses on whether a tool can correctly mark a program as (not) CT. More recently, Geimer et al. [20] summarize and analyze the methodologies of over 30 CT analysis tools. The authors further select five promising tools from them and evaluate them with several cryptographic implementations. For each tool, the benchmark reports the analysis time and the number of reported CT violations. Based on the results, the work qualitatively discusses the limitations of existing tools.

User Studies. Jancar et al. [15] interview cryptographic developers to understand how CT analysis tools are adopted in practice and to investigate the reasons that prevent developers from using them. According to the study, the most popular tool is CT-grind [6], which was developed in 2010. Furthermore, they show that poor usability and inadequate documentation are among the primary barriers preventing developers from trying more CT analysis tools. Fourné et al. [21] conduct a user study to assess the usability of six CT analysis tools with a set of small programs. They show that dynamic analysis tools generally have better usability.

Work that proposes new tools. It is common to compare a new CT analysis tool with existing tools when publishing it [7], [28], [8], [29], [30], [13], [31], [32]. They typically benchmark tools on real-world cryptographic implementations and demonstrate the ability to discover known vulnerabilities such as CVEs [28], [29], [8], [13], [31]. Some of them conduct false-positive and false-negative analyses based on case studies [28], [29], [30], [13], [31], [32]. Furthermore, analysis time efficiency [7], [28], [8], [29], [30], [13], [31], [32] and, in some cases, memory consumption [28], [31] would also be used as performance metrics. One limitation in them is that they often have their unique experiment setups, such as building their own benchmark sets, comparison baselines, and metrics when comparing with existing tools that have already been evaluated under other configurations. Therefore, it complicates the universal comparison of tools across the literature.

Identified Issues. By reviewing the aforementioned works, we summarize the issues they identified into three categories: *Operational Performance*, *Usability*, and *Detection Quality*. Operational performance refers to the runtime resource consumption, such as time efficiency [7], [28], [8], [29], [30], [13], [31], [32] and memory usage [28], [31]. Tools that have higher runtime resource requirements would be less likely to be adopted in practice [15], [21]. This category also evaluates the scalability of a tool by evaluating whether it can analyze large and complex programs and how many resources would be consumed. The scalability issue has been explicitly reported [15], and empirical studies frequently observe resource exhaustion, such as timeouts, when analyzing complex programs [20], [30], [29], [18].

Usability is consistently identified as a major barrier [20], [15], [21] for adopting a tool in industry. Prior work reports difficulties in installation and configuration, as well as challenges in interpreting tool outputs. Some works [20], [15] suggest that report interpretability can be improved by reducing excessive information in reports.

Compared to the operational performance and usability, detection quality is more frequently discussed by previous works. Specifically, several limitations of existing tools have been identified in terms of missing CT violations and re-

porting false positives. First, not covering all three CT programming rules, particularly variable-time instructions [40], would miss CT violations [20]. Second, not tracking implicit flow leakages (e.g., secret indirectly propagates to variables through a conditional branch) [20]. Third, dynamic analysis tools that compare the traces of different executions may miss a CT violation if no divergence is triggered by sufficient inputs [20]. Fourth, dynamic analysis tools that present traces in a tree structure may face analysis saturation to obscure violations by other violations at a higher tree level, resulting in violations at lower levels not being reported [20], [28]. Fifth, not supporting all instructions, such as SIMD instructions, may also lead to miss CT violations [20]. Sixth, dynamic analysis may report false positives if it cannot deal with internal randomness that does not depend on secrets [8]. Seventh, static analysis may report false positives due to over-approximation [13]. For example, all variables in a structure might be considered as secret even though only one of them is secret. Finally, dynamic-taint and symbolic execution tools may report false positives on dead branches that are never executed in practice [17], [18].

3.4. Composite CT Analysis Pipelines

Since 2025, several CT analysis pipelines, which combine multiple analysis tools, have been proposed to maximize the analysis performance [17], [18], [19]. Interestingly, they converge on similar choices of CT analysis tools: an (optional) statistical test followed by a taint-based dynamic analyzer and a symbolic static analyzer. The motivation is intuitive: dynamic analysis cheaply surfaces violations but is unsound, while symbolic execution might cover leakages that are missed by unsound tools [17]. However, whether this combination does maximize the CT analysis performance has not been quantitatively evaluated.

4. Benchmark Design

We present the threat model, architecture, workflow, ground-truth construction process, and metrics used by *AutoBench-CT* for quantitative evaluation of CT analysis tools.

4.1. Threat Model

Our threat model assumes sequential execution, leaving the analysis of speculative and out-of-order execution to future work. Specifically, we prohibit secret-dependent branch conditions, memory addresses, and operands of variable-time instructions. Since the set of variable-time instructions is platform-dependent, we tailor our threat model to our experimental platform, an Intel i7-14700K. We treat integer division instructions (`DIV/IDIV`) as variable-time, but exclude multiplication instructions (`MUL/IMUL/MULX`), based on Intel's official list of data-operand-independent timing instructions [62], empirical measurements [63], [64], and recent attacks [41]. Other platforms may require a different set of variable-time instructions, in which case the benchmark's ground truth must be re-established accordingly. CT analysis tools may model a different set of variable-time instructions than our threat model. We handle such discrepancies as follows. If a tool fails to detect a violation involving an instruction covered by our threat model, we count it as a false negative (FN). Conversely, if a tool reports a violation involving an instruction outside our threat model, we exclude that report from the evaluation, as it falls outside the scope of our ground truth.

4.2. Design Overview

The design of *AutoBench-CT* is inspired by prior well-established benchmarking efforts, Magma [22] and SV-COMP [65], [25], which systematically evaluate software testing tools against established ground truths. At its core, *AutoBench-CT* adopts a similar *modular design*: tasks (program under test and target ground-truth labels), tools, and the orchestration logic are kept independent of one another and communicate only through small unified interfaces. As a result, any tool in the benchmark can do analysis on any tasks out of the box, and adding a new task or a new tool requires no modification to the others or to the core framework. Given a set of tasks and a set of CT analysis tools, *AutoBench-CT* produces a quantitative comparison report across the tools along a fixed set of metrics, using its built-in per-task ground-truth labels of CT violations as the reference. The concrete component breakdown, interface definitions, execution workflow, and extension procedure are provided in Appendix A.

Secret annotations. To ensure every tool analyzes exactly the same secret for every task, *AutoBench-CT*'s modular design decouples the secret annotation from the program under test. Specifically, each program under test is wrapped as a task using a unified interface that includes: the secret `SECRET_TYPE secret`, which is also the only input with changing values for the program under test; the `read_secret(SECRET_TYPE secret, ...)` function to load the concrete values of the secret from a local file; and the `task(SECRET_TYPE secret, SECRET_LEN)` function that executes the program under test. `secret` here can be just a partial region of the secret key variable for the program under test, as a secret key variable may contain both a secret region and a public region [18]. We expose the secret region as the individual variable `secret` and use `memcpy` to propagate the `secret` into the secret key within `task(...)`. In this case, a tool can be integrated into our benchmark as long as it can mark the entire variable `secret` as a secret. To analyze a task, a tool implements its own driver by invoking an abstract task via `read_secret(...)` and `task(...)`. Before `task(...)`, the tool uses its own annotation approach to specify the entire `SECRET_TYPE secret` variable as the secret. We show a concrete example of how this mechanism works with CT-grind in Listing 2.

4.3. Ground Truth Generation

As demonstrated in Section 2, fine-grained, security-relevant evaluation of CT analysis tools requires a per-violation label. However, no reliable automated method exists for exhaustively enumerating all secret-dependent violations in a binary. We therefore bootstrap ground truth from the union of violations reported by the benchmarked tools and refine it through cross-tool agreement and independent human review. We strictly follow the same posture adopted by mature ground-truth-driven security benchmarks, Magma [22] and SV-COMP [65], [25], and case study [27] to provide the most defensible labels we can, and state precisely what they cover.

CT violation representation. *AutoBench-CT* compiles each task once into a static library that every tool then links against with its own Task Driver, so the program-under-test is byte-identical across tools. We unify every reported violation to source-line granularity; if a tool only reports binary addresses, we map them back with `addr2line` (GCC) or `llvm-addr2line` (Clang). Each violation is encoded as `FILE.c LINE:TYPE`, with `TYPE` $\in$ {`A` (secret-dependent memory access), `B` (conditional branch), `V` (variable-time operation)}; e.g., `AES.c 108:A`. Source-line unification is necessary for two reasons. First, binary addresses are not comparable across tools. Although every tool links the same crypto archive, each tool wraps the same task in its own driver (e.g., ctgrind injects `VALGRIND_MAKE_MEM_UNDEFINED`), which differs in code size and runtime dependencies. Thus, the linker places the identical crypto code at different absolute addresses. Consequently, a single source-level leak may appear at `0x401234` under CT-grind but `0x80489ab` under Binsec. Second, developers patch source code regardless of where a tool emits the violation, so source-line labels also match the natural debugging workflow.

CT violation classification. We classify each reported violation as a true positive (TP), false positive (FP), or ignored positive (IG). A violation is a TP only if it ❶ is present in the compiled binary, ❷ violates the CT policy, and ❸ belongs to the program under test. Criterion ❶ is critical from a security standpoint: a secret-dependent ternary operator in C may or may not survive as a branch after compilation, and an analysis warning that the compiler has already eliminated is not a real side channel. However, one may consider it a potential CT violation and would like to mark it as a TP. It is feasible to do so when manually labeling the ground truth, but we only count violations remaining in the compiled binary as TPs. A violation is an FP if it does not violate the CT policy or does not appear in the binary, and an IG if it lies outside the program under test (e.g., CT-grind sometimes flags lines inside the tool itself) or if its source-code-level location is uninterpretable (i.e., `??`, `0`, or `unknown`). Unless specified otherwise, we exclude IGs from all metrics. We do not optimize the ground-truth set for different policies when reporting CT violations. We discuss its impact in Section 7.

Two-pass ground truth generation. We label each source line in two passes: a coarse automatic pass followed by a fine manual correction. In the coarse pass, *AutoBench-CT* votes across all configured tools and marks a violation as a TP if at least t tools agree on it (default t=2), as an FP otherwise, and as an IG when the location is uninterpretable. Accuracy grows with the number of participating tools, but two failure modes remain: the vote can wrongly promote a violation that $\geq t$ tools jointly mis-report, and wrongly reject one that fewer than t tools detect. Because neither prior work provides such ground truth nor exhaustively enumerating all secret-dependent violations is feasible, we made our best effort to balance automation and accuracy. However, in this work, we introduce a manual correction on each pre-classified label to maximize the accuracy of our results.

Soundness and Completeness. There are two limitations in our ground truths. First, our completeness is bounded by the union of participating tools. A violation that none of them detects can only be recovered by exhaustive code review, infeasible for complex programs. Second, our precision is bounded by human inspection, which can be error-prone. Both limits are shared to ground-truth-driven evaluation: Magma [22] integrates published CVEs as ground truths only, and SV-COMP [65], [25] relies on sustained manual updates by research groups. To mitigate the first, we verified that none of the violations that match known CVEs [66] in our selected tasks were missed (i.e., being reported by at least one evaluated tool). To mitigate the second, we follow a cross-checking strategy similar to prior work [27]. We invited two independent participants: one expert in cryptography, side-channel analysis, and data-flow analysis, and one in cryptography and data-flow analysis, to review the labels separately, with remaining disagreements resolved by consensus.

4.4. Evaluation Metrics

We design the quantitative metrics based on the discussion and findings, mostly empirical, presented in prior work [15], [20], [21], [26], [18], [28], [8], [30], [13], including both benchmark studies and works that propose new tools. Specifically, we quantitatively assess CT analysis tools from three aspects: *Operational Cost*, *Detection Quality*, and *Usability*.

4.4.1. Operational Cost. This metric evaluates the runtime behavior of a CT analysis tool, focusing on its computing resource usage and scalability.

Computing resource usage. *AutoBench-CT* evaluates computing resource usage in two dimensions: *time efficiency* and *memory consumption*. Time efficiency, consistent with prior benchmarking work [20], is measured by *AutoBench-CT* 's core execution engines as the wall-clock time required from start to complete the analysis for each tool–task pair. Memory consumption, which was measured [28], [31] as a metric and also complained about by users [15], is measured by the peak RAM usage during execution and the size of generated output files occupying disk space.

Scalability. While there is no universally applicable metric to directly measure the scalability [20], we combine Cyclomatic complexity [67] and the number of def-use memory operations to reflect the complexity of the program under test. Cyclomatic complexity is a well-established metric for measuring control-flow path complexity [68], [69], [70], while the number of def-use memory operations captures memory path complexity, which is particularly relevant for CT analysis. We compute both metrics using the Control-Flow Graph derived by `angr` [71], and compute the final complexity score by:

$$Score = log_2(Cyclomatic) + log_2(MemOps) \quad (1)$$

Based on the score, we categorize tasks into four complexity classes using K-Means clustering. The classification is not fixed and will be updated whenever a program is added or removed. For each complexity class, we report the minimum, maximum, and median analysis time and peak RAM usage, along with the timeout and memory-exhaustion rates for each tool. Together, these metrics constitute our scalability evaluation.

4.4.2. Detection Quality. This metric evaluates how reliably a CT analysis tool detects TPs and avoids FPs. With ground-truth labels, we quantify the detection quality in four dimensions: *precision and relative recall*, *classification-flipping rate*, *uniqueness*, and *potential issue diagnosis*.

Precision and relative recall. We compute the precision with $TP/(TP + FP)$ and relative recall with $TP/(TP + FN)$. Our recall is relative because the number of false negatives (FNs) depends on how many TPs can be detected by all tools, which may not necessarily cover all in-theory TPs.

Classification-flipping rate. This metric measures the percentage of tasks whose classifications were flipped due to the tools' limitations (i.e., a tool reports at least one violation on a CT task or a tool reports no violation on a non-CT task). It indicates how reliably a tool's reporting behavior aligns with the ground-truth CT status of a task.

Uniqueness. This metric captures the number of TPs that can only be reported by one tool but not by the others. It is used to evaluate the unique advantage of a tool in detecting CT violations.

Potential issue diagnosis. This metric helps tool developers quickly localize why a TP is missed, or a FP is raised, which otherwise requires substantial manual inspection of tool reports [20], [18], [26]. We build a Capture-the-Flag-style challenge from a curated set of CT violations, organized by the known tool-issue categories collected in Section 3.3. For each category, we pick representative violations and flag the category as a potential issue for a tool if it reports none of them (FN categories), reports at least one (FP categories), or fails to complete the relevant tasks (FP categories). We further classify each flagged issue as *Classification-Flipping*, which can change the tool's overall CT/non-CT Classification on a task (e.g., missing variable-time divisions), or *Count-Reducing*, which only changes the number of detected TPs (e.g., missing implicit flows, which must accompany a preceding branch violation). Because one underlying defect may surface across multiple categories, the metric serves as an indicator of potential issues requiring post-analysis to confirm.

4.4.3. Usability. Apart from runtime behavior and detection quality, usability is also an important factor in evaluating a tool. Since usability is a relatively subjective metric and is often evaluated through user studies, we only focus on quantifiable factors that have been discussed by previous works [15], [21], [20]. Specifically, we quantify the *time to first output (TTFO)* and *report interpretability*.

TTFO. This metric is defined as the elapsed time until the first entry appears in the violation report during analysis runtime. The intuition is that it would be more efficient if a cryptographic developer does not need to wait for the full report (especially when analyses are subject to timeouts) before he/she can start investigating the issue [20].

Report interpretability. We quantify the report interpretability by measuring both the percentage of hard-to-analyze violations (i.e., violations reported with an unknown source code location) and the average number of report lines per violation. This metric serves as an indicator of how much manual effort could be required to investigate the issue based on tool reports.

4.5. Composite Tool Pipeline Evaluation

Based on the metrics defined for each individual tool, we design the *detection coverage* and *operational practicality* metrics to evaluate the performance of a composite tool pipeline. To further evaluate how each tool contributes to the overall performance of such a composite tool pipeline, we design an ablation study metric *Marginal Detection Yield*.

Detection coverage. This metric counts the overall number of TPs and FPs detected by the composite tool pipeline.

Operational Practicality. This metric evaluates whether the pipeline finishes within a certain resource budget. While those tools can be run in parallel, we consider the pipeline to be run sequentially for simplicity. Therefore, we sum up the runtime but simply pick the peak RAM usage across all tools in the pipeline and compare it with a predefined resource budget.

Marginal Detection Yield (MDY). This metric evaluates the contribution to detection coverage of each tool. For a candidate tool T added to a baseline pipeline $\mathcal{S}$, MDY is the number of TPs that T contributes *and* $\mathcal{S}$ misses, optionally normalized by additional cost (peak wall-clock time and peak RAM). Formally,

$$\mathrm{MDY}(T \mid \mathcal{S}) \;=\; |\,\mathrm{TP}(T) \setminus \bigcup_{S \in \mathcal{S}} \mathrm{TP}(S)\,|. \quad (2)$$

MDY directly measures whether including T is justifiable: a tool with $\mathrm{MDY} = 0$ adds no detection power, regardless of how it performs in isolation.

TABLE 2: CT analysis tools evaluated in this work.

Tool	Pub./Upd.	Technique	Version	Setup	Doc	Analysis Granularity (level)
Abacus [29]	2021/2022	Dyn. symbolic exec on traces	Git `f35c039`	repo		Address range (byte-level)
Binsec [9]	2020/2026	Sta. symbolic	v0.10.0	doc		Address range (byte-level)
CT-grind [6]	2010/2026	Dyn. taint	Valgrind 3.23+patch [40]	doc		Address range (byte-level)
DATA [8]	2018/2018	Dyn. trace cmp + statistics test	Git `9d467ed`	doc		Input files (bit-level difference)
Microwalk [28]	2022/2025	Dyn. trace cmp	Git `v3.1.1`	doc		Input files (bit-level difference)
Pitchfork [42]	2020/2020	Sta. symbolic	Git `8a3d325`	repo		Address range (byte-level)

5. Benchmark Implementation

We instantiate *AutoBench-CT* with 16 cryptographic implementations and six binary-level CT analysis tools, and prepare ground truth using the procedure of Section 4.3.

5.1. Cryptographic Implementations Under Test

We select 16 representative implementations spanning symmetric, asymmetric, and post-quantum cryptography, taken from BearSSL 0.6 (AES-CBC, AES-GCM, CHASSE, CHACHA, CTAES-CBC, CTAES-GCM, ECDSA, RSA-OAEP), MbedTLS 3.6.3 (AES-CBC, CHACHA, ECDSA, RSA-OAEP), PQClean commit 2cc6471 (ML-KEM key generation, ML-KEM decapsulation, HQC decapsulation), and Ryde signing from a related work [18] which collects an earlier version of Ryde's official implementation that contains a representative CT violation (code link). We always treat the secret key as the "secret". All AES and CHACHA implementations perform decryption, while all RSA and ECDSA implementations perform signing. Each implementation is compiled with `Clang 18.1.8` under all four combinations of {`-O0`, `-O3`}×{`-m32`, `-m64`}, yielding 64 tasks in total.

We intentionally include implementations with known CT violations, including ones covered by recent CVEs (e.g., CVE-2025-52473, CVE-2024-36405, CVE-2022-0778) and prior literature [40], [72]; implementations whose name starts with "CT" are intended to be CT. Across all 64 tasks, we manually labeled 2227 reported violations. We count each leaky source line once per task, so the same line appearing in two binaries of the same implementation under different flags is counted twice (e.g., BearSSL AES-CBC has 36 violations in each `-O0` binary and 32 in each `-O3` binary, totaling $36{\cdot}2 + 32{\cdot}2 = 136$). Unlike prior benchmarks [26], [20] that fix one compile configuration, we vary both optimization level and architecture, which alters binary structure even for identical source code.

5.2. CT Analysis Tools Under Test

We evaluate six binary-level CT analysis tools, summarized in Table 2. They were chosen because they are either popular among industrial developers [15] or frequently used as baselines in the literature [21], [20], [26], [17], [18], [19]. The six tools span both dynamic and static families and four sub-techniques (Section 3.2): dynamic taint (CT-grind), dynamic trace comparison (DATA, Microwalk), dynamic symbolic execution (Abacus), and static symbolic execution (Binsec, Pitchfork-angr). We additionally apply the Kyberslash patch [40] to CT-grind to enable variable-time-instruction (e.g., divisions) checks, and treat DATA as two analysis targets: DATA-P1, which performs the first-phase dynamic trace comparison to detect CT violations; and DATA-P2, which performs the second-phase dynamic fix-vs-random dynamic tests to filter out "unleaked" violations detected in DATA-P1. CT-grind, which is based on Valgrind [16], and Binsec, which is based on BINSEC [73], are being actively maintained, while the remaining four tools are academic prototypes We set up all tools following their public instructions. Details for each tool are in Appendix D. As shown in Table 2, all tools satisfy *AutoBench-CT*'s granularity requirement of marking at least an entire variable as the secret (Section 4.2).

Compared with prior benchmarks [26], [20] and recent composite pipelines [17], [18], [19], we share all binary-level CT analysis tools, while excluding non-binary-level (e.g., Flowtracker [11]) and statistical-only (e.g., Dudect [10]) approaches to maintain fair binary-level ground truth. We focus on binary analysis because compiler-induced CT violations [74], [17], [75], [72] are only observable at the binary level. We also exclude MemSan (working on LLVM IR) from the main evaluation because of unfair comparison. However, we discuss it in Section 6.5 individually.

5.3. Evaluation Configuration

We ran all experiments on an Intel i7-14700K machine (64 GB RAM, Ubuntu 22.04). Each implementation was compiled with `Clang 18.1.8` using its upstream library's default flags, with `-O0`/`-O3` and `-m32`/`-m64` appended to override the optimization level and target architecture as the task required, and `-g` added to retain debug information. For tools that report violations only at binary addresses (Binsec, DATA, Microwalk, Pitchfork), we mapped reports back to source lines with `llvm-addr2line`.

While prior work [8] reports that 10 inputs suffice for the execution coverage of complex cryptographic programs, we provided 16 inputs for each task (same as prior works [20], [28]) to be conservative. We also measured per-task line coverage for 16 inputs and observed 90% of the program's lines with all core cryptographic computation (e.g., AES round operations, big-number modular exponentiation, and polynomial arithmetic) across all evaluated tasks have been covered. Uncovered lines correspond to public-flag-gated branches unreachable in our setup. For instance, since we always invoke ML-KEM decapsulation, the flag for matrix generation is always set to the transposed-matrix branch, making the original-matrix branch unreachable.

Typically, we kept the default settings for all tools as soon as they were installed. However, we increased Binsec's `sse-depth` to 10^6 (consistent with prior works [20], [17]), and further `sse-jump-enum` to 10^6, to avoid too-early path-cut limited by its too small default settings (1,000 and 3, respectively). We further replaced selected `glibc` routines (`memcpy/memcmp/memset/memmove`) with external implementations to avoid Binsec consistently stalling

TABLE 3: Operational Cost

	Abacus	Binsec	CT-grind	DATA-P1	DATA-P2	Microwalk	Pitchfork
Time (s)	1.4–⧗ (15)	**0.1**–⧗ (1.0K)	4.8–**710** (**6.1**)	2.2–1.7K (11)	2.1–1.7K (18)	1.0–⧗ (**6.1**)	1.4–⧗ (17)
RAM (MB)	**40**–▦ (**65**)	64–▦ (863)	138–**288** (180)	396–▦ (2.8K)	405–▦ (2.8K)	134–9.3K (160)	195–▦ (263)
Disk (MB)	0.6–2.5K (5.4)	0.8–**1.7** (0.9)	**0.1**–147 (**0.5**)	1.4–36K (41)	1.4–36K (47)	0.8–40K (44)	0.8–18 (1.0)
Exit Normally	26/32 (81%)	30/64 (47%)	**64/64** (**100%**)	30/32 (94%)	30/32 (94%)	29/32 (91%)	42/64 (66%)

For Time, RAM, and Disk, the upper line is min-max and the lower line is median. ⧗ refers to timeout (3600 s), and ▦ refers to exceeded the RAM limit (50 GB). 'Exit Normally' refers to the number of tasks that the tool self-terminates without crashes, and its percentage. Due to compatibility issues, Microwalk and DATA are evaluated on 64-bit tasks only; Abacus on 32-bit only; the others on all 64 tasks. Numbers $\geq$1000 are shown to two significant figures with the 'K' suffix. Full per-task-complexity-class statistics is in Appendix B.

there. For fairness, violations at those replaced functions are ignored. We did not observe such default-setting issues in similar symbolic tools (Abacus and Pitchfork). As a result, we keep them as default, same as works [20], [17].

We set a 3,600 s wall-clock timeout (consistent with work [20] and twice that of work [18]) and a 50 GB peak RAM limit for each task to ensure stable execution on our machine.

5.4. Ground Truth Preparation

After setting up the experiment environment, we followed the procedure of Section 4.3 with cross-comparison threshold $t = 2$ and then manually verified every classification. We independently labeled all 2227 reported violations and resolved disagreements jointly; only 12 disagreements arose, all in RSA due to complex data flow.

Automatic cross-comparison initially classified 735 TPs, 1447 FPs, and 45 IGs; after manual verification, the final ground truth contains 2058 TPs, 18 FPs, and 151 IGs. The large FP-to-TP correction stems from the 32-bit tasks: only CT-grind and three symbolic tools support `-m32`, but all symbolic tools suffered from incomplete analysis in most of these tasks. Thus, violations reported only by CT-grind were initially miscounted as FPs by cross-comparison and were corrected manually.

We summarize the characterization of all 16 selected cryptographic primitives in Table 8 in Appendix C.

6. Evaluation

In this section, we report the quantitative results and answer the three research questions in Section 1.

6.1. Operational Cost Evaluation

Table 3 summarizes operational cost. For architectural compatibility, we restrict the evaluation of Microwalk and DATA to 64-bit tasks and Abacus to 32-bit tasks. CT-grind is the only tool that completes every task within the resource budget, with median runtime in seconds and median peak RAM 180 MB, although its Class-4 runtime spikes to 710 s on Ryde-1f. DATA and Microwalk each fail two Class-4 Ryde tasks, and Microwalk additionally fails HQC `-O0` at Class 3. All of them suffer from RAM exhaustion or timeout while collecting Intel-Pin traces, but Microwalk crashes early once the trace size hits a threshold, keeping its RAM records finite. All three symbolic tools (Abacus, Binsec, Pitchfork) hit both timeouts and memory limits, consistent with prior findings on path explosion or solver saturation (e.g., RSA GCD) [20], [18]. Among them, Abacus exhausts resources less often than Binsec and Pitchfork.

The per-tool extremes show two distinct failure shapes. Symbolic tools are bimodal, where many tasks can finish in seconds, but a substantial fraction exhaust resources. Intel-Pin trace collectors (Microwalk, DATA, Abacus) suffer from large traces tens of GBs of disk and exceeding 50 GB of RAM consumption. CT-grind alone keeps a low and stable footprint across all complexity classes. According to Table 8 and Table 3, for the symbolic tools, task scale alone does not explain symbolic tools' failures. The bimodality of failures comes from whether they face an analysis barrier. Without those barriers, they can finish the analysis quickly. Otherwise, they exhaust the analysis budget. For instance, Binsec times out at a Class-2 task (BEARSSL AES-GCM `-O0 -m64`) due to solver saturation. However, it successfully completes a Class-3 task (MBEDTLS CHACHA `-O3 -m64`), whose binary size, number of machine instructions, and program complexity are higher than BEARSSL AES-GCM. In contrast, dynamic tools are solely influenced by the task scale.

Overall, CT-grind delivers the most robust and scalable operational performance, combining low resource consumption with a 100% completion rate across all program classes.

6.2. Detection Quality Evaluation

Table 4 summarizes detection qualities. No tool catches every TP, and only CT-grind and the trace-comparison tools (DATA, Microwalk) reach non-trivial recall.

32-bit tasks. CT-grind detects all TPs except the 36 violations in MBEDTLS AES-CBC, due to being unable to track explicit flows of secret-indexed memory accesses [20], [13], and adds two FPs. However, all those missed TPs are detected by, and only by, Binsec. Among the three symbolic tools (the only others supporting 32-bit), Binsec and Abacus reach 0.16 relative recall and Pitchfork only 0.05. All three suffer from resource exhaustion, path cutting, and unsupported symbols (e.g., SIMD in Abacus) [20], [17], [18]. Pitchfork additionally terminates each path on the first violation found, which is its designed behavior (and is consistent with the settings in work [17]). We attempt to override it to let it continue, but it can stall even at a single branch (e.g., enumerating 2^{256} paths, for one branch per iteration over 256 iterations, in Listing 1, line 4).

64-bit tasks. CT-grind again leads with 0.71 recall, 392 unique TPs, and four FPs. The big drop in recall is due to

TABLE 4: Detection Quality Statistics

	32-bit							64-bit						
	Abacus	Binsec	CT-grind	DATA-P1	DATA-P2	Microwalk	Pitchfork	Abacus	Binsec	CT-grind	DATA-P1	DATA-P2	Microwalk	Pitchfork
Expected TPs	876	876	876	876	876	876	876	1182	1182	1182	1182	1182	1182	1182
Detected TPs	140	139	**840**	-	-	-	44	-	139	**840**	739	277	380	32
Detected FPs	**0**	**0**	2	-	-	-	**0**	-	**0**	4	12	4	4	**0**
Detected URs	0	36	**661**	-	-	-	0	-	0	**392**	165	0	31	0
Precision	**1.000**	**1.000**	0.998	-	-	-	**1.000**	-	**1.000**	0.995	0.984	0.986	0.990	**1.000**
Recall	0.160	0.159	**0.959**	-	-	-	0.050	-	0.118	**0.711**	0.625	0.234	0.321	0.027
Flip CT → Non-CT	**0/0/15**	**0/0/15**	1/0/15	-	-	-	**0/0/15**	-	**0/0/14**	**0/0/14**	2/0/14	2/0/14	2/0/14	**0/0/14**
Flip Non-CT → CT	11/11/17	7/7/17	**0/0/17**	-	-	-	6/6/17	-	8/8/18	**0/0/18**	4/2/18	6/2/18	5/2/18	9/9/18

Classification flipping rows are represented as `[c/k/n]`. `c`: number of tasks being flipped. `k`: number of incompleted tasks in `c`. `n`: expected number of tasks being CT/Non-CT. An 'incomplete task' means the tool crashed, exhausted resources on it, or the analysis was early terminated due to path cutoff (Symbolic tools only).

violations reported by DATA and Microwalk, which do not run on 32-bit. Specifically, DATA-P1 (0.63, 165 URs) and Microwalk (0.32, 31 URs) complement CT-grind by catching implicit-flow violations that it misses (e.g., MbedTLS RSA-OAEP `-O0`). However, trace comparison still fails on tasks without trace divergence (e.g., BearSSL RSA-OAEP `-O0`). Binsec and Pitchfork report no unique TPs, but inherit their 32-bit struggles. Notably, DATA's differential-slicing-based alignment [76] exhibits the same saturation effect previously documented for Microwalk [28]: each tool occasionally misses violations the other reports (e.g., MbedTLS RSA-OAEP `-O0`).

False positives. Only the symbolic tools produce no FPs. Trace-based tools cannot distinguish violations driven by secret-independent randomness (e.g., ML-KEM key generation internally produces a random secret whose traces differ per run). DATA-P2's statistical post-processing neither eliminates such FPs (PQClean ML-KEM Keygen `-O0`) nor preserves all TPs in DATA-P1 (it removes TPs from PQClean HQC-DEC `-O3`). Taint analysis is immune to randomness FPs but flags dead secret-dependent branches (e.g., Ryde sign `-O0`) [18].

Classification flipping. Although no single tool catches every TP, only CT-grind never flips a task's classification from non-CT to CT (i.e., reports no violations on a non-CT task). The reason is that its main FN classes are count-reducing issues (e.g., implicit flows) rather than classification-flipping issues (e.g., unsupporting variable-time instructions). All three symbolic tools flip $\sim 50\%$ of non-CT tasks as CT, primarily because their analyses do not complete (early-cut paths or resource exhaustion). DATA and Microwalk each flip about five non-CT tasks, from missing trace divergence or from resource exhaustion. However, DATA-P2's statistical filter is particularly damaging. It flips two further non-CT tasks to CT by removing TPs, making it unreliable as a CT-violation detector. In reverse (CT → non-CT), only CT-grind, DATA, and Microwalk flip tasks, all due to FPs.

Impact of compilation. An interesting finding is that the compilation procedure may affect how a CT violation is reported. Specifically, we find that compiler optimizations may alter binary-to-source mappings, leading to fewer reported violations. For example, the number of violations reported in `Clang` compiled BearSSL AES-CBC `-O3` is fewer than `-O0`. We further test the binary of BearSSL AES-CBC and AES-GCM compiled with GCC 11.4.0 and find that mapping binary location to the source line may also generate

TABLE 5: Potential Issue Diagnosis

Type		Issue	Abacus	Binsec	CT-grind	DATA-P1	DATA-P2	Microwalk	Pitchfork
FN	C-F	Secret dependent memory access	✓	✓	✓	✓	✓	✓	✓
		Secret dependent conditional branch	✓	✓	✓	✓	✓	✓	✓
		Secret dependent variant-time instructions	?	✓	✓	?	?	?	?
		SIMD support	?	✓	✓	✓	✓	✓	✓
		32-bit support	✓	✓	✓	?	?	?	✓
		64-bit support	?	✓	✓	✓	✓	✓	✓
		No divergence on traces	?	?	✓	?	?	?	✓
		Uncompleted analysis due to scalability	?	?	✓	?	?	?	?
		Violations induced by compiler optimizations	?	?	✓	✓	✓	✓	✓
	C-R	Explict flow by memory access	?	✓	?	✓	✓	✓	✓
		Implicit flow	?	?	?	✓	✓	✓	?
		Analysis saturation	?	?	✓	?	?	?	?
FP	C-F	Secret-independent violations	✓	✓	✓	?	?	?	?
		Over approximation	✓	✓	✓	✓	✓	✓	?
		Dead branch	✓	✓	?	?	?	?	✓

We use ✓ for a tool that detected at least one labelled violation in FN issues or at least one task has been finished but none of the labelled violations reported in FP issues. Otherwise, it is represented as ?, as an indicator of a potential issue for manual confirmation. C-F for Classification-Flipping and C-R for Count-Reducing.

incorrect source locations under non-O0 optimization levels. It is different from the well-known compiler-induced CT violations [74], [17], [75], [72], which may increase the number of TPs. Instead, we find that compilation optimization may cause troubles to locate the CT violation in the source code when investigating or fixing the violation. We do not count them as FPs or FNs, as the tools correctly identified them in the binary, and the mapping mechanism is orthogonal to the design of CT analysis tools.

Automatic issue diagnosis. Table 5 lists automatically diagnosed issues. The results show that *AutoBench-CT* recovers documented limitations [20], [28], [21], [15], [8], [17], [18] of CT-grind, Microwalk, and DATA, and discovers two new ones: DATA still suffers from analysis saturation, and DATA-P2's statistical post-processing still produces FPs. For the three symbolic tools, most flagged issues stem from incomplete analyses. To further diagnose them, one needs to make manual efforts to either modify the program under test or update the tool.

Variable-time instructions. Only CT-grind and Binsec support variable-time instructions. When this option is enabled, CT-grind assumes and only assumes all integer divisions as variable-time. In contrast, Binsec additionally considers integer multiplications, but exposes the assumption of the set as a configurable feature to allow users to customize

TABLE 6: Usability

	Abacus	Binsec	CT-grind	DATA-P1	DATA-P2	Microwalk	Pitchfork
Uninter-pretability	0/296 (0.0%)	110/720 (15.3%)	240/46232 (0.5%)	522/1774 (29.4%)	53/544 (9.7%)	0/46494 (0.0%)	0/6427 (0.0%)
Lines per Violation	4.3	2.5	198.3	279.3	2199.1	6.4	61.2
TTFO (s) min-max	0.0 ⧗	0.0 ⧗	0.1 0.1	0.1 1665	0.1 1575	0.2 ⧗	0.0 0.1

Uninterpretability counts the number of hard-to-analyze violations over all violations the tool reports. ⧗ for timeout.

which instructions are considered variable-time for different microarchitectures. To match the threat model on Intel i7-14700k (Section 4.1), we configure Binsec to check only integer divisions. Consequently, no entries in Table 4 or Table 5 are affected by the divergence of different variable-time set assumptions, as none of the tools assumes a variable-time set beyond the threat model's. Table 5 also correctly shows CT-grind and Binsec as the only two tools supporting variable-time checks.

Overall, CT-grind achieves the highest precision, recall, and number of unique findings among all evaluated tools.

6.3. Usability Evaluation

Table 6 summarizes usability. CT-grind generates the largest raw volume (46,232 violations vs 2,227 unique violations), since it re-reports the same line once per distinct call-context execution [20]. Microwalk and Pitchfork show similar repetition.

Source-level interpretability. Abacus, Microwalk, and Pitchfork map every violation back to a source line, and CT-grind reaches 0.5% uninterpretability. DATA-P1 (29.4%) and Binsec (15.3%) are far weaker. This appears at both `-O0` and `-O3`, so it is not purely an optimization artifact. Additionally, Binsec can produce a report with all reported violations that are unintepretable (e.g., it returns six binary-address violations on HQC DEC `-O3`, none of which `llvm-addr2line` resolves). Such reports can confuse developers in determining whether the program is actual CT.

Report verbosity. CT-grind and DATA generate hundreds to thousands of lines per violation, while it is 61 for Pitchfork generates and < 10 for the rest. Note that CT-grind supports two report formats: plaintext and `xml`. Such a volume is driven by its XML format used for automated processing, while its plaintext format averages only 24 lines per violation.

Runtime feedback. Only CT-grind, Binsec, and Pitchfork stream results during execution; the others report only at completion. Pitchfork continuously outputs the current execution point, while CT-grind and Binsec stay silent until the first violation. For Binsec, this can mean the developer waits up to the $3{,}600$ s timeout with no feedback. However, CT-grind usually completes in seconds, so the limitation is rarely observed in practice.

Overall, CT-grind offers the most balanced usability: high interpretability, runtime feedback, and a plaintext-report option that controls verbosity.

6.4. Ablation Study of Composite Pipelines

All three 2025 composite-pipeline works [17], [18], [19] pair taint-based dynamic analyzer(s) with a symbolic static analyzer. We use the per-task data from §6.2 to test whether this taint+symbolic recipe is empirically optimal. For fairness, Abacus is compared on 32-bit tasks only, while DATA and Microwalk are compared on 64-bit tasks only.

Detection coverage. CT-grind reports hundreds of unique TPs across every level (Table 4), so any maximum-coverage pipeline must contain it. Starting from CT-grind, Abacus contributes nothing ($\mathrm{MDY}(\mathrm{Abacus} \mid \mathrm{CT\text{-}grind}) = 0$); Binsec contributes 36 and 44 TPs in 32- and 64-bit tasks, respectively; Pitchfork contributes 8 unique TPs in 64-bit tasks only, but they disappear once a CT-Binsec pipeline is used ($\mathrm{MDY}(\mathrm{Pitchfork} \mid \{\mathrm{CT\text{-}grind}, \mathrm{Binsec}\}) = 0$). In contrast, Binsec still contributes 72 TPs in a CT-Pitchfork pipeline ($\mathrm{MDY}(\mathrm{Binsec} \mid \{\mathrm{CT\text{-}grind}, \mathrm{Pitchfork}\}) = 72$). It justifies the "CT-Binsec" recipe in prior works [17], [18] on coverage grounds, and may also explain why Binsec frequently became the choice of the symbolic tool in recent works [20], [18], [15], [21]. However, CT-Binsec stops at a 64-bit recall of 0.748, while adding a dynamic trace-comparison tool changes the picture: CT-DATA reaches 0.974, and CT-DATA-Microwalk reaches 1.000. Critically, once a trace-comparison tool is in the pipeline, $\mathrm{MDY}(\mathrm{Binsec} \mid \{\mathrm{CT\text{-}grind}, \mathrm{DATA}\}) = \mathrm{MDY}(\mathrm{Binsec} \mid \{\mathrm{CT\text{-}grind}, \mathrm{Microwalk}\}) = 0$. Trace comparison tools subsume Binsec's explicit-flow advantage and close CT-grind's implicit-flow blind spot (the only two FN classes known for CT-grind), while CT-grind in turn covers the no-divergence and scalability failures of trace-comparison. Conversely, symbolic execution adds nothing to them.

Operational practicality. CT-Binsec consumes a median of 1196 s and 1.7 GB peak RAM yet finishes only $15/32$ tasks within budget at recall 0.748. CT-DATA finishes $30/32$ tasks at recall 0.974 in 18 s and 2.8 GB; CT-Microwalk finishes $29/32$ at recall 0.826 in 11 s and 0.2 GB. Both taint+trace-comparison pipelines therefore deliver substantially higher recall and complete roughly twice as many tasks in two orders of magnitude less time. Consequently, CT-Binsec has no competitive advantage. Extending CT-DATA with Microwalk closes the residual 0.026 recall gap, but at $+22\%$ median time and $+45\%$ disk overhead, and one fewer task within budget ($30/32$ vs $29/32$). Furthermore, this gap is an artifact of the count-reducing trace saturation issue [28] in both DATA and Microwalk: a violation will re-surface once its parent is fixed. We therefore recommend CT-DATA as the empirically optimal pipeline for CT violation detection.

Verification coverage and the residual role of symbolic tools. A purely dynamic pipeline cannot prove that a program is CT, which is why the security argument for keeping symbolic tools in the recipe holds. Among the six evaluated tools, only Binsec and Pitchfork are sound by construction;

Abacus, despite using symbolic execution, runs on Intel-Pin traces from concrete inputs and inherits the soundness gap of dynamic coverage. In practice, the proof capability of the two sound tools is much narrower than the recipe assumes. Of the 29 tasks intended to be CT, Binsec certifies 16 (55%) and Pitchfork certifies 19 (66%). All the certified tasks are AES/CHACHA implementations at Classes 1–3, but none (Pitchfork proves one) of the post-quantum implementations at Classes 3–4. Similarly, Binsec in work [18] completes zero post-quantum cryptography analysis. Thus, slotting either tool into a composite pipeline does not unlock proofs on the code, unless users first manually slice the program into small fragments and re-annotate secrets per fragment.

However, only symbolic execution may certify the absence of CT violations under bounded soundness. What our results challenge is the operational claim that symbolic tools should be a drop-in pipeline component. **Overall, given the strong detection coverage and operational efficiency of the dynamic pipeline ("CT-DATA"), we recommend using it first to eliminate reported violations. Symbolic tools (e.g., Binsec) can be reserved for a final bounded-sound verification stage (rather than violation finding) when the dynamic pipeline is silent, thereby minimizing symbolic cost while retaining proof capability.**

6.5. Comparison with a Non-Binary-Level Tool

To probe the gap between binary and non-binary level analyzers, we additionally run MemSan [34] (Clang 18), a dynamic taint analyzer on LLVM-IR that uses a similar conceptual approach as CT-grind. MemSan supports only 64-bit binaries. It executes the compiled binary concretely, but its taint instrumentation is injected at LLVM-IR compile time, so violations are reported at LLVM-IR granularity. Because it does not distinguish `A`/`B`/`V` violation types, we relax the violation match to "same source line" for this comparison, under which our 64-bit ground truth reduces to 1055 TPs and 16 FPs. While it is unfair to use binary-level labels to evaluate MemSan, we aim to disclose whether it can identify violations that remain in executables.

Detection. MemSan completes all 64-bit tasks faster than CT-grind (median 1.5 s, 0.09 GB peak RAM, excluding IR-generation time). It reports 267 violations, of which 150 match labeled TPs and zero FPs/IGs. The other 117 do not appear in our ground truth. Compared with the expected 1055 TPs, MemSan reports only a small subset of secret-derived violations. Our analysis suggests that it is due to an aggressive reporting policy: MemSan shadow propagation completes through every IR instruction, but its runtime emitter deduplicates warnings sharing the same provenance chain, so many downstream operations (e.g., AES round key derived from the secret key) are consolidated into the earliest parent violation rather than re-reported individually. It also underestimates the gap between LLVM-IR and the final binary (e.g., HQC-DEC `-O3` has only three compiler-induced violations, but MemSan reports zero violations there). Furthermore, MemSan cannot detect variant-time-instruction violations.

Tool-induced "mirage" violations. The 117 (44% of all reported) MemSan reports without a ground-truth match are noise from MemSan's own checks (induced by `-fsanitize` instrumentation) but not real violations. For instance, MemSan reports a violation at HQC DEC -O0 vector.c line 82 (Link), which calls function `reduce()`, with binary address `0xd0ffc`. By disassembling, this address is a sanitizer-inserted `call __msan_warning_with_origin` which is MSan's runtime warning emitter. The preceding three instructions (`cmp ...; je ...; mov ...,%edi`) form the shadow-byte guard that `-fsanitize=memory` injects to decide whether to invoke the emitter. Together, all four instructions are sanitizer-only, and the source line 82 compiles to a CT Barrett arithmetic, triggering no violation in any other tool when compiled without the sanitizer flags. We confirm that all 117 reports land on sanitizer-inserted warning calls, not on user code. Those reports flip the classification of eight of 14 tasks from CT to non-CT.

As noted in Section 4.3, CT-grind sometimes flags lines inside the tool itself as well. Excluding repeated reports but including IGs, we found 76 such cases (4%) among its 1797 reported violations. Different from MemSan, CT-grind never modifies the binary. Instead, it runs the program inside Valgrind, which may intercept `glibc` calls by its own implementations and report any violations found in them. However, the user impact is very different. CT-grind's mirages can be filtered directly from the source-level report: they always sit inside a small, fixed set of Valgrind-internal functions whose labels start with `vg_`. In contrast, the offending instructions are sanitizer's code with no source-level counterpart, yet MemSan still attributes each one to a valid source line in the user's program. Unless inspecting each reported binary address, they are indistinguishable from real violations in source-level reports.

6.6. Answer to RQ1: Which Tool Achieves the Strongest Overall Performance

CT-grind dominates in operational performance and detection quality, with low and stable resource use and the highest precision and relative recall among the six. Although it produces verbose reports, its report interpretability remains acceptable: fewer than 1% of reported violations are uninterpretable, and it provides intermediate feedback during execution. This quantitatively explains the user-study finding [15] that CT-grind remains the most-used tool in industry: it offers the best balance of scalability, accuracy, and usability among all evaluated tools.

6.7. Answer to RQ2: Recurring Limitations

We identify four mostly recurring limitation patterns that appear across the six evaluated tools plus MemSan.

Exhaustive resource consumption on path exploration. All tools except CT-grind and MemSan hit a resource exhaustion limit. Intel-Pin trace collectors (Microwalk,

DATA, Abacus) consume up to 50 GB of memory or disk, while symbolic tools (Binsec, Pitchfork, and the dynamic-symbolic Abacus) suffer from path explosion and are prone to incomplete analysis.

Tool-induced violations. Tools may unintentionally modify analysis targets and introduce violations that do not exist in the original program. CT-grind introduces 4% of them, while the number for MemSan is 44% (which flips the classification of eight of 14 tasks from CT to non-CT). These tool-induced violations can misclassify a program's CT/non-CT status and confuse developers in issue localization.

Precise data-flow tracking. Four tools explicitly exhibit data-flow tracking limitations, while the other three are under-determinant. Fully capturing both explicit and implicit data flows is challenging, leading to missed violations (best recall is only 0.71 in 64-bit tasks). Moreover, incomplete data-flow information hinders tools from reliably determining whether detected violations are truly secret-dependent.

Report interpretability and runtime feedback. Four tools generate massive output (tens of thousands of violations, or hundreds of lines per violation), three tools produce uninterpretable locations (∼10% for two of them, 29% for one), and no tool tells the user, during execution, which fragments have already been verified CT.

Because no single tool captures all TPs (0.71 best on 64-bit tasks and 0.96 best on 32-bit tasks), we strongly recommend combining tools together as discussed in RQ3.

6.8. Answer to RQ3: How should CT Analysis Tools be Combined in Practice?

The dynamic taint+symbolic recipe shared by three recent composite-pipeline works [18], [17], [19] is not the empirically optimal composition. Our ablation study shows that the dynamic taint+trace comparison pipeline ("CT-DATA") reaches 0.974 recall and remains practically deployable (30/32 tasks within budget). Symbolic execution retains a unique role for verification of small, dynamically clean fragments, rather than for improving detection coverage.

6.9. Recommendations for Future Works

For future tool designers:

Ensuring the scalability first. Among the evaluated tools, only CT-grind consistently completes all analyses within the time and memory budgets. Even so, it already experiences noticeable pressure when analyzing large post-quantum implementations such as Ryde-1f. As those cryptography will be deployed in the near future, scalability will become an essential requirement for practical CT verification.

Being careful of modifying analysis targets. Modifying the analysis target can bring tool-induced violations. While our experiments only identified such an issue in Valgrind and LLVM-IR instrumentations, we remind tool designers to be careful if similar modifications exist.

Improving report interpretability. Readable and actionable reports substantially reduce the effort required for fixing CT violations. Tools can improve usability by consolidating duplicate messages, reporting concise violation summaries, and providing additional contextual information, such as call stacks, related addresses, or reporting the range spanned by the nearest instructions that do map to source code, for violations that are difficult to map back to source code to narrow down the human searching space.

Advancing CT analysis through specialization. Our experiments suggest that combining complementary tools can improve security assurance without requiring a single tool to handle all classes of CT violations. We thus encourage future work on lightweight, specialized analyzers that address specific limitations and integrate into composite analysis pipelines. Such an ecosystem would enable the community to build progressively stronger CT assurance through complementary advances across tools and techniques.

Evaluating tools with comprehensive quantitative metrics. We encourage future tool papers to adopt shared benchmarks with comprehensive quantitative metrics that are of concern to the community (e.g., those in *AutoBench-CT*), turning competing claims into measurable progress and giving users a quantitative basis for tool selection.

For future cryptography programmers:

Using composite analysis pipelines. No single tool detects all CT violations, but combining complementary tools can substantially improve coverage. In particular, "CT-DATA" achieves much higher detection coverage than any individual tool with manageable additional cost. Beyond violation detection, users can balance bounded-sound verification against practical analysis cost based on their security needs.

7. Limitations

Our work aims to improve the quantitative evaluation of CT analysis tools and provide practical guidance for tool and methodology selection, rather than declaring a universally best solution. We focus on violation detection in x86 binaries compiled from C, and do not evaluate dimensions such as formal CT proofs or applicability to other platforms and languages (e.g., Microwalk extension Easywalk [31] for JavaScript). In addition, our results may not reflect the maximum achievable performance of tools under extensive expert-driven tuning (e.g., Binsec in work [18]). To reduce evaluation bias and improve reproducibility, all tools are configured strictly according to their official public documentation (Table 2) with minimal manual intervention. Furthermore, not reporting some violations may be due to the specific reporting policy rather than the ability to detect them. For instance, implicit-flow violations are arguable as reported violations [20], [11], and tools like MemSan may have aggressive filtering policies to avoid reporting them. Instead of justifying which policy is correct, we marked all violations that indeed violate the CT policy as TPs, quantified the cost of ignoring them, and left the options to users. Finally, the ground-truth set for `V` labels is a microarchitecture-dependent choice. In future work, *AutoBench-CT* will support automatically filtering `V`-related

reports by checking each reported binary location against an architecture-specific `V` instruction list.

8. Conclusion

We presented *AutoBench-CT*, a unified benchmark with per-violation ground truth and quantitative metrics for CT analysis tools. We quantitatively showed that CT-grind, a tool published in 2010, still outperforms other tools by achieving the best across all three evaluation categories. We raised six recommendations for future tool designers and users, and recommended the taint+trace comparison (CT-DATA) pipeline for CT violation detections.

References

[1] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *USENIX Security*, 2014, pp. 719–732.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *CT-RSA*. Springer, 2006, pp. 1–20.

[3] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO*, 1996, pp. 104–113.

[4] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[5] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying {Constant-Time} implementations," in *USENIX Security*, 2016, pp. 53–70.

[6] A. Langley, "ctgrind: Checking that functions are constant time with valgrind," https://github.com/agl/ctgrind, 2010, accessed: 2025-04-10.

[7] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *ACSAC*, 2018, pp. 161–173.

[8] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "{DATA}–differential address trace analysis: Finding address-based {Side-Channels} in binaries," in *USENIX Security*, 2018, pp. 603–620.

[9] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *IEEE S&P*. IEEE, 2020, pp. 1021–1038.

[10] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1697–1702.

[11] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 110–120.

[12] Q. Zhou, S. Dang, and D. Zhang, "{CtChecker}: A precise, sound and efficient static analysis for constant-time programming," in *ECOOP*, 2024, pp. 46–1.

[13] Z. Zhang and G. Barthe, "Ct-llvm: Automatic large-scale constant-time analysis," *Cryptology ePrint Archive*, 2025.

[14] CRoCS, "Ct-tools: Constant-timeness verification tools," https://crocs-muni.github.io/ct-tools/, accessed: 2026.01.27.

[15] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""they're not that hard to mitigate": What cryptographic library developers think about timing attacks," in *IEEE S&P*. IEEE, 2022, pp. 632–649.

[16] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[17] L. Gerlach, R. Pietsch, and M. Schwarz, "Do compilers break constant-time guarantees?" in *International Conference on Financial Cryptography and Data Security*. Springer, 2025, pp. 327–344.

[18] O. Adjonyo, S. Bardin, E. Bellini, G. N. Dione, M. F. A. Ameen, R. Merget, F. Recoules, and Y. Sellami, "Systematic timing leakage analysis of NIST PQDSS candidates: Tooling and lessons learned," in *DIMVA*, 2026.

[19] S. Deng, Z. Liu, S. Wang, and Y. Zhang, "An empirical study measuring in-the-wild cryptographic microarchitectural side-channel patches," in *ACM CCS*, 2025, pp. 648–662.

[20] A. Geimer, M. Vergnolle, F. Recoules, L.-A. Daniel, S. Bardin, and C. Maurice, "A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries," in *ACM CCS*, 2023, pp. 1690–1704.

[21] M. Fourné, D. D. A. Braga, J. Jancar, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""these results must be false": A usability evaluation of constant-time analysis tools," in *USENIX Security*, 2024, pp. 6705–6722.

[22] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[23] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1393–1403.

[24] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *ACM ISSTA*, 2021, pp. 662–665.

[25] SoSy-Lab, "Sv-benchmarks: Benchmark suite for software verification," https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, 2025, accessed: 2026.01.19.

[26] A. C. Lopes and D. F. Aranha, "Benchmarking tools for verification of constant-time execution," *SBSEG. Bento Goncalves, Brazil: SBC*, pp. 716–726, 2017.

[27] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, "Toward effective secure code reviews: an empirical study of security-related coding weaknesses," *Empirical Software Engineering*, vol. 29, no. 4, p. 88, 2024.

[28] J. Wichelmann, F. Sieck, A. Pätschke, and T. Eisenbarth, "Microwalk-ci: Practical side-channel analysis for javascript applications," in *ACM CCS*, 2022, pp. 2915–2929.

[29] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *IEEE/ACM ICSE*. IEEE, 2021, pp. 797–809.

[30] L. Cai, F. Song, and T. Chen, "Towards efficient verification of constant-time cryptographic implementations," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1019–1042, 2024.

[31] I. Fayolle, J. Wichelmann, A. Köhl, W. Rudametkin, T. Eisenbarth, and C. Maurice, "Semi-automated and easily interpretable side-channel analysis for modern javascript," in *International Conference on Cryptology and Network Security*. Springer, 2024, pp. 25–46.

[32] M. Dunsche, M. Maehren, N. Erinola, R. Merget, N. Bissantz, J. Somorovsky, and J. Schwenk, "With great power come great side channels: Statistical timing {Side-Channel} analyses with bounded type-1 errors," in *USENIX Security*, 2024, pp. 6687–6704.

[33] S. Cauligi, C. Disselkoen, K. V. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *ACM PLDI*, 2020, pp. 913–926.

[34] Clang Team, "Memorysanitizer," https://releases.llvm.org/18.1.8/tools/clang/docs/MemorySanitizer.html, accessed: 2026.01.27.

[35] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "{BunnyHop}: Exploiting the instruction prefetcher," in *USENIX Security*, 2023, pp. 7321–7337.

[36] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *EuroS&P*. IEEE, 2019, pp. 142–157.

[37] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[38] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *CHES*, 2010, pp. 110–124.

[39] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE S&P*, 2015, pp. 623–639.

[40] D. J. Bernstein, K. Bhargavan, S. Bhasin, A. Chattopadhyay, T. K. Chia, M. J. Kannwischer, F. Kiefer, T. Paiva, P. Ravi, and G. Tamvada, "Kyberslash: Exploiting secret-dependent division timings in kyber implementations," *Cryptology ePrint Archive*, 2024.

[41] R. L. Schröder, S. Gast, and Q. Guo, "Divide and surrender: Exploiting variable division instruction timing in HQC key recovery attacks," in *USENIX Security*, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/schr%C3%B6der

[42] C. Disselkoen, S. Cauligi, D. Tullsen, and D. Stefan, "Finding and eliminating timing side-channels in crypto code with pitchfork," in *TECHCON*, 2020.

[43] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *ACM CCS*, 2017, pp. 859–874.

[44] S. He, M. Emmi, and G. Ciocarlie, "ct-fuzz: Fuzzing for timing leaks," in *IEEE ICST*. IEEE, 2020, pp. 466–471.

[45] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *USENIX Security*, 2015, pp. 897–912.

[46] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *ACM CCS*, 2018, pp. 131–145.

[47] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," *Science of Computer Programming*, vol. 78, no. 7, pp. 796–812, 2013.

[48] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *ACM CCS*, 2017, pp. 875–890.

[49] K. Athanasiou, B. Cook, M. Emmi, C. MacCárthaigh, D. Schwartz-Narbonne, and S. Tasiran, "Sidetrail: Verifying time-balancing of cryptosystems," in *VSTTE 2018*. Springer, 2018, pp. 215–228.

[50] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM TISSEC*, vol. 18, no. 1, pp. 1–32, 2015.

[51] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *ACM PLDI*, 2017, pp. 406–421.

[52] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation," *Journal of Computer Security*, vol. 27, no. 1, pp. 137–163, 2019.

[53] A. Schaub, O. Rioul, and S. Guilley, "Stanalyzer: A simple static analysis tool for detecting cache-timing leakages," in *17th International Workshop on Cryptographic Architectures Embedded in Logic Devices (CryptArchi 2019)*, 2019.

[54] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying {Cache-Based} side channels through {Secret-Augmented} abstract interpretation," in *USENIX Security*, 2019, pp. 657–674.

[55] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "{CacheD}: Identifying {Cache-Based} timing channels in production software," in *USENIX Security*, 2017, pp. 235–252.

[56] C. Sung, B. Paulsen, and C. Wang, "Canal: a cache timing analysis framework via llvm transformation," in *ACM/IEEE ASE*, 2018, pp. 904–907.

[57] S. Chattopadhyay and A. Roychoudhury, "Symbolic verification of cache side-channel freedom," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2812–2823, 2018.

[58] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *IEEE S&P*. IEEE, 2019, pp. 505–521.

[59] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. R. Butler, and D. J. Tian, "Encider: detecting timing and cache side channels in sgx enclaves and cryptographic apis," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1577–1595, 2022.

[60] K. Jiang, Y. Bao, S. Wang, Z. Liu, and T. Zhang, "Cache refinement type for side-channel detection of cryptographic software," in *ACM CCS*, 2022, pp. 1583–1597.

[61] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *ACM CCS*, 2021, pp. 715–733.

[62] Intel Corporation, "Data operand independent timing instructions," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html, 2026, document ID 768547. Accessed: 2026.09.11.

[63] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs," https://www.agner.org/optimize/instruction_tables.pdf, 2025, technical University of Denmark. Last updated 2025-09-20. Accessed: 2026.09.11.

[64] A. Abel and J. Reineke, "uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 673–686.

[65] D. Beyer, "State of the art in software verification and witness validation: Sv-comp 2024," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 299–329.

[66] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[67] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*. US Department of Commerce, Technology Administration, National Institute of ..., 1996, vol. 500, no. 235.

[68] D. Guaman, P. Sarmiento, L. Barba-Guamán, P. Cabrera, and L. Enciso, "Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis," in *Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering*, 2017, pp. 171–175.

[69] Google OSS-Fuzz Team, "Fuzzing introspection," https://github.com/ossf/fuzz-introspector, 2026, accessed 2026.01.24.

[70] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: An fmri study," in *IEEE/ACM ICSE*. IEEE, 2021, pp. 524–536.

[71] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *IEEE S&P*. IEEE, 2016, pp. 138–157.

[72] Z. Lai, R. Zhang, Z. Zhang, J. Hermelink, M. Schwarz, V.-T. Pham, and U. Parampalli, "You only decapsulate once: Ciphertext-independent single-trace passive side-channel attacks on hqc," *Cryptology ePrint Archive*, 2025.

[73] BINSEC Team, "BINSEC: Binary-level analysis platform," https://github.com/binsec/binsec, accessed: 2026-10-11.

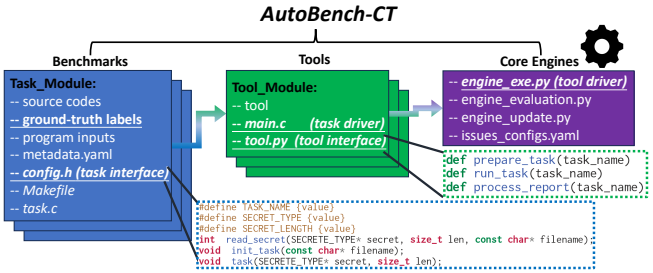


Figure 1: *AutoBench-CT* Architecture

Listing 2 An example of *AutoBench-CT*'s Task Driver for CT-grind.

```
int main(int argc, char *argv[]){
  SECRET_TYPE secret[SECRET_LENGTH];
  int ret = read_secret(secret,
            SECRET_LENGTH, argv[1]);
  init_task(argv[1]);
  VALGRIND_MAKE_MEM_UNDEFINED(secret,
    sizeof(SECRET_TYPE)*SECRET_LENGTH);
  task(secret, SECRET_LENGTH);
  return 0;
}
```


[74] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking bad: How compilers break constant-time˜ implementations," *arXiv preprint arXiv:2410.13489*, 2024.

[75] T. Pornin, "Constant-time code: The pessimist case," *Cryptology ePrint Archive*, 2025.

[76] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications," in *IEEE S&P*. IEEE, 2011, pp. 347–362.

[77] M. Neikes. (2019) Timecop. Accessed: 2026.01.27. [Online]. Available: https://www.post-apocalyptic-crypto.org/timecop

[78] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

Appendix A. *AutoBench-CT* Framework Details

This appendix provides the full concrete realization of *AutoBench-CT*— its modular architecture, the two execution workflows, and the procedure for extending the benchmark with new programs or new tools — whose logic is summarized in Section 4.2.

A.1. Architecture

Inspired by well-established benchmarks for evaluating software testing and verification tools, such as Magma [22] and SV-COMP [65], [25], *AutoBench-CT* also has a modular design and ground-truth labels. It is decoupled into three components to facilitate customization and future extension. We present a design overview in Figure 1. It uses multiple core engines to manage all tests and decouples the configuration of constant-time analysis tools and programs under test into two parts.

Benchmarks. As shown in Figure 1, we refer to the first component as *Benchmarks*. It contains all programs under test to be analyzed by constant-time analysis tools. Each program is structured as an independent *task* module that contains the program's source code, ground truth labels, program inputs that were prepared in advance, a `metadata.yaml` file that stores basic program characteristics (e.g., Cyclomatic complexity [67]), and a Makefile. The program under test is implemented in `task.c` and wrapped using a unified interface defined in `config.h`. Specifically, the core program under test is implemented in `task(...)`, whose arguments specify the secret value and its length. The function `read_secret(...)` loads the prepared secret input for `task(...)` from the `program inputs` directory, while `init_task(...)` initializes non-secret data required by `task(...)` (e.g., a fixed public key). Finally, the wrapped program is precompiled into a static shared library. This design allows constant-time analysis tools to always start analysis from `task(...)` and to swap programs simply by changing the linked library when compiling the executable. Using a static shared library also ensures the program-under-test binary is consistent across different tools.

Tools. We refer to the second component, which manages all constant-time analysis tools under test, as *Tools*. For each tool, *AutoBench-CT* prepares an individual *tool* module that contains the tool's source code, the collected raw reports, and a task driver (`main.c`). We present an example of the task driver for CT-grind in Listing 2. It first declares and loads the secret value, initializes the task, invokes `VALGRIND_MAKE_MEM_UNDEFINED(...)` to taint the secret for CT-grind tracking, and then executes the program under test. A unified *Tool Interface* is implemented in `tool.py` to invoke the tool under test. By feeding a task name (i.e., program under test), the three functions shown in Figure 1, initialize the task, invoke the tool to analyze it through the corresponding task driver, and then parse the raw analysis report into an expected format and save it to a file. Similarly to the *Task Interface*, *Tool Interface* enables *AutoBench-CT* to invoke all supported tools consistently, regardless of their original command-line syntax, configuration requirements, or execution procedures. It also eliminates the need for modifying the core benchmarking framework when integrating new tools.

Core Engine. We build the last component, referred to as *Core Engine*, with three Python scripts to manage the entire workflow of *AutoBench-CT*. The update engine (`engine_update.py`) manages the metadata computation and ground truth label generation of a task. The execution engine (`engine_exe.py`) implements a tool driver that, by feeding a tool name and a task name, invokes a tool module to analyze a task module, records the runtime resource consumptions (e.g., time and peak RAM usage), and formats reported violation representations. The evaluation engine (`engine_evaluation.py`) computes the

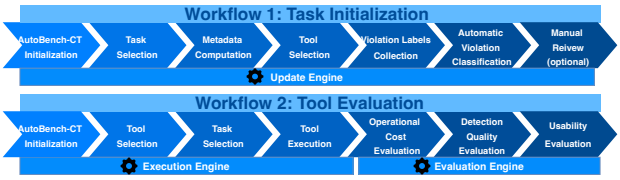


Figure 2: *AutoBench-CT* Workflow 1 and 2

evaluation results for metrics defined in Section 4.4.

A.2. Workflow

AutoBench-CT operates through two major workflows: *Task Initialization* (Figure 2) and *Tool Evaluation* (Figure 2). Both workflows begin with an initialization phase, where *AutoBench-CT* configures all required environment paths and compiles tools and tasks into a ready-to-use state.

The *Task Initialization* workflow prepares a task for analysis. It first computes the metadata (including the complexity) of a program under test using the update engine. By specifying which tools participate in ground-truth generation, it then produces ground-truth labels through automatic cross-comparison, which can later be manually updated.

The *Tool Evaluation* workflow assesses the selected tool(s) using the specified task(s), based on the evaluation metrics defined in Section 4.4. Each tool-task pair is executed sequentially while the execution engine monitors runtime behavior. Once all reports are collected, the evaluation engine is invoked to finalize the evaluation and generate the overall benchmark report.

A.3. Extending *AutoBench-CT*

Because of the modular design, the tools and programs under test can be easily removed or added. To integrate a new benchmark task, one should create a task module that is similar to the example in Figure 1. This involves adding the necessary source code, implementing the *Task Interface*, and configuring the inputs and Makefile. Once the module is ready, one should run the Task Initialization workflow to compute metadata and generate ground-truth labels. Similarly, adding a new CT-analysis tool requires creating a tool module, incorporating the tool's source code, and implementing the task driver along with the *Tool Interface*. Then, one should execute the Tool Evaluation workflow to evaluate the tool on selected tasks. To update ground-truth labels for violations reported by the new tool, one simply re-executes the Task Initialization workflow by adding the new tool in step four: Tool Selection. Finally, the Capture-the-Flag–style potential issue diagnosis can also be modified by updating the `issues_configs.yaml` to add, remove, or update challenges in a modular manner.

Appendix B. Operational Performance by Each Task Complexity Class (Table 7)

Appendix C. Per-Primitive Benchmark Characterization (Table 8)

Appendix D. Constant-Time Analysis Tools: Detailed Setup

This appendix expands Table 2 with the full description and setup procedure for each evaluated tool.

Abacus. Abacus [29] (latest HEAD) is a dynamic symbolic-execution tool that differs from Binsec in two ways: it captures execution traces with Intel Pin and then performs trace-oriented symbolic execution directly over x86 instructions, and it detects violations by solving CT constraints with a specialized probabilistic algorithm rather than SMT solvers. Abacus requires DWARF 4 debug info, so we add `-gdwarf-4` to its compilation flags; it also only supports 32-bit binaries (`-m32`). We set it up following the tutorial.

Binsec. Binsec [9] (0.10.0) performs relational symbolic execution on binaries, checking whether a tainted secret can cause divergence in memory access or control flow between two executions. Variable-time checks were added in 0.10.0 as an experimental feature by detecting tainted operands of multiplication and division. Binsec offers *bounded* soundness: the analysis is sound up to the number of executed instructions. We increased Binsec's `sse-depth` to 10^6 (consistent with prior works [20], [17]), and further `sse-jump-enum`, to avoid too-early path-cut limited by its too small default settings (1,000 and 3, respectively), and replaced selected `glibc` routines (`memcpy`, `memcmp`, `memset`, `memmove`) with external implementations in the task driver to avoid analysis stalls. Violations at these replaced symbols are ignored. We set it up following the tutorial.

CT-grind. CT-grind [6] (Valgrind 3.23), the first published constant-time analysis tool [15], performs dynamic taint analysis by marking the secret address as uninitialized and leveraging Valgrind [16] to track its propagation. Because Valgrind is widely deployed for memory-safety checking, CT-grind integrates naturally into developer workflows and is the most popular CT tool in the industry [15]. By default, it covers only memory accesses and control-flow transfers; we applied a recent unofficial patch [40] to extend coverage to variable-time operations. We set it up following the tutorial. Note that the original CT-grind patch to Valgrind is no longer necessary, and prior work [18] may use Timecop [77] (another tool that applies exactly the same CT-grind approach but simply patches Valgrind functions with another name). In this paper, we choose "CT-grind" as the canonical name for this taint-based dynamic analysis approach via Valgrind, given its historical priority. Since Timecop uses the same analytical approach, our evaluations transfer to it directly. Note that work [17] also uses the name "CT-grind".

TABLE 7: Operational Cost (full distribution per complexity class).

		All	Class 1 (0.0-14.7)	Class 2 (14.7-17.9)	Class 3 (17.9-23.6)	Class 4 (23.6-inf)
Time (s)	abacus	1.4-⧗(15.4)	1.4-21.4 (3.8)	1.6-41.0 (2.6)	1.5-⧗(21.9)	68.8-⧗(1429.4)
	binsec	**0.1**-⧗(1015.3)	**0.1**-**0.3** (**0.1**)	**0.1**-⧗(**0.3**)	**0.1**-⧗(3557.8)	**0.1**-⧗(⧗)
	ctgrind	4.8-**709.7** (**6.1**)	4.8-5.9 (5.0)	4.9-**6.7** (5.9)	5.0-**37.3** (**6.7**)	9.0-**709.7** (**10.8**)
	DATA-P1	2.2-1701.4 (11.0)	2.2-9.7 (2.3)	2.2-37.0 (2.3)	2.3-700.2 (13.4)	6.3-1701.4 (301.8)
	DATA-P2	2.1-1670.8 (17.8)	2.1-16.9 (2.1)	2.1-43.8 (2.3)	2.2-1023.1 (18.6)	6.3-1670.8 (314.4)
	microwalk	1.0-⧗(**6.1**)	1.0-1.0 (1.0)	1.0-71.5 (1.5)	1.1-773.5 (39.5)	36.4-⧗(358.1)
	pitchfork	1.4-⧗(17.1)	1.5-16.9 (2.7)	1.5-38.3 (7.2)	1.4-⧗(523.2)	3.1-⧗(2591.8)
RAM (MB)	abacus	**40.1**-▬(**65.1**)	**40.2**-**40.4** (**40.3**)	**40.1**-**67.1** (**46.2**)	**40.5**-▬(1255.5)	**48.8**-▬(6344.1)
	binsec	63.8-▬(863.4)	68.6-94.5 (92.7)	63.8-9737.9 (94.0)	91.2-▬(3096.0)	84.9-▬(7752.1)
	ctgrind	137.7-**287.7** (180.0)	139.8-183.2 (182.8)	137.7-188.0 (168.0)	138.4-**287.7** (**181.3**)	140.5-**224.2** (**177.5**)
	DATA-P1	396.1-▬(2837.4)	397.5-2718.9 (398.4)	396.1-3120.0 (407.3)	420.0-46624.2 (2872.8)	3041.5-▬(3598.7)
	DATA-P2	405.2-▬(2836.6)	405.9-2797.6 (415.5)	405.2-3115.8 (416.2)	416.2-40352.4 (2862.5)	3325.8-▬(11311.5)
	microwalk	133.5-9264.5 (159.7)	133.5-146.9 (142.5)	144.3-2090.2 (153.8)	151.6-7158.7 (273.8)	171.7-9264.5 (1541.0)
	pitchfork	195.0-▬(263.4)	211.8-312.3 (225.6)	207.4-349.2 (249.8)	195.0-▬(1699.4)	217.1-▬(12649.5)
Disk (MB)	abacus	0.6-2474.7 (5.4)	0.6-**1.0** (0.9)	1.6-6.9 (3.6)	1.2-2100.4 (95.4)	479.3-2474.7 (1499.9)
	binsec	0.8-**1.7** (0.9)	0.8-1.1 (0.9)	0.8-**1.1** (0.9)	0.8-**1.4** (1.1)	1.7-**1.7** (**1.7**)
	ctgrind	**0.1**-146.5 (**0.5**)	**0.1**-1.2 (**0.1**)	**0.1**-29.6 (**0.1**)	**0.1**-73.5 (**0.3**)	**1.1**-146.5 (87.2)
	DATA-P1	1.4-35881.4 (40.7)	1.4-5.9 (2.0)	1.7-571.1 (7.9)	9.6-35881.4 (202.4)	952.5-27377.2 (11672.8)
	DATA-P2	1.4-35881.4 (46.5)	1.4-7.0 (2.0)	1.7-577.6 (7.9)	9.6-35881.4 (205.8)	1076.4-26819.8 (10858.3)
	microwalk	0.8-39573.5 (43.5)	0.8-6.2 (1.6)	1.9-3517.3 (23.7)	5.8-39573.5 (1027.8)	1920.7-32489.4 (9853.2)
	pitchfork	0.8-17.5 (1.0)	0.8-**1.0** (1.0)	0.8-1.2 (1.0)	0.9-17.5 (1.2)	1.8-3.1 (2.4)
Self-terminated without crash	abacus	26/32 (81.2%)	3/3 (100.0%)	9/9 (100.0%)	11/14 (78.6%)	3/6 (50.0%)
	binsec	30/64 (46.9%)	7/7 (100.0%)	14/19 (73.7%)	8/26 (30.8%)	1/12 (8.3%)
	ctgrind	64/64 (100.0%)	7/7 (100.0%)	19/19 (100.0%)	26/26 (100.0%)	12/12 (100.0%)
	DATA-P1	30/32 (93.8%)	4/4 (100.0%)	10/10 (100.0%)	12/12 (100.0%)	4/6 (66.7%)
	DATA-P2	30/32 (93.8%)	4/4 (100.0%)	10/10 (100.0%)	12/12 (100.0%)	4/6 (66.7%)
	microwalk	29/32 (90.6%)	4/4 (100.0%)	10/10 (100.0%)	11/12 (91.7%)	4/6 (66.7%)
	pitchfork	42/64 (65.6%)	7/7 (100.0%)	19/19 (100.0%)	14/26 (53.8%)	2/12 (16.7%)

Time, RAM, and Disk data are represented as [min]-[max] (median). ⧗ refers to timeout (3600 s), and ▬ refers to exceeded the RAM limit (50 GB).

TABLE 8: Characterization of the 16 cryptographic primitives, each built into four binaries (`m32`/`m64` × `-O0`/`-O3`).

Primitive	LoC	Binary (KB)	Insn (K)	CC ($\log_2$) (Class)	Mem ($\log_2$) (Class)	TPs
Complexity Class 1–2						
BR-CHA-SSE	1308	29–36	0.7–2.0	5.5(1)-6.8(1)	6.8(1)-9.3(2)	0
BR-AES-CBC	1364	30–33	1.0–1.5	5.5(1)-7.1(1)	6.9(1)-8.6(2)	32–36
BR-CHACHA	1240	30–33	0.8–1.9	5.5(1)-6.8(1)	7.2(1)-9.0(2)	0
BR-CTAES-CBC	1497	37–43	1.5–2.2	6.4(1)-7.2(1)	8.2(1)-9.2(2)	0
Complexity Class 2						
BR-AES-GCM	1700	44–47	1.6–2.9	6.8(1)-7.6(1)	8.1(1)-9.6(2)	32–36
BR-CTAES-GCM	1580	45–68	1.8–4.2	6.8(1)-6.9(1)	8.3(1)-10.0(2)	0
Complexity Class 2–3						
BR-RSA-OAEP	1950	75–88	3.8–5.4	8.4(2)-8.4(2)	8.7(2)-10.4(3)	60–63
MB-CHACHA	2456	64–75	2.6–4.4	8.4(2)-8.5(2)	8.8(2)-10.3(3)	0
Complexity Class 3						
MB-AES-CBC	3241	72–92	3.9–7.0	8.6(2)-8.8(2)	9.2(2)-10.8(3)	27
PQ-MLKEM-KGen	1798	84–167	5.0–9.6	8.6(2)-9.6(2)	10.2(3)-11.3(3)	0
PQ-MLKEM-Dec	1798	85–168	5.0–9.6	8.7(2)-9.6(2)	10.3(3)-11.3(3)	0–2
BR-ECDSA	2802	106–299	6.6–12.1	9.5(2)-9.5(2)	10.2(3)-11.3(3)	2–8
PQ-HQC-Dec	1840	109–227	7.2–13.0	9.5(2)-9.8(2)	10.7(3)-11.7(3)	0–3
Complexity Class 4						
Ryde-Sign	2999	391–957	36.6–92.0	11.0(3)-12.1(3)	12.5(3)-14.4(4)	0–9
MB-ECDSA	50833	1288–1678	80.8–139.4	13.9(4)-14.2(4)	13.7(4)-15.2(4)	104–192
MB-RSA-OAEP	50833	1288–1682	80.9–139.5	13.9(4)-14.2(4)	13.7(4)-15.2(4)	156–239

`[num]-[num]`: the min-max value of this field; `BR`: BearSSL; `MB`: mbedTLS; `PQ`: PQClean. `LoC`: non-comment lines of source code, over the library sources the binary links in (by 'DWARF'). `Insn`: number of machine instructions (by 'objdump'). `CC`/`Mem`: $\log_2$ of Cyclomatic complexity and of the number of memory operations (by 'angr'). `TPs`: number of TPs.

DATA. DATA [8] (commit `9d467ed`) is a 2018 dynamic tool that runs in two phases: phase 1 collects execution traces under different inputs via Intel Pin [78] and flags any divergence as a CT violation; phase 2 applies fixed-vs-random statistical tests to distinguish secret-dependent violations from noise. We evaluate the two phases separately as DATA-P1 and DATA-P2. We set it up following the usage guide.

Microwalk. Microwalk [28] (v3.1.1) is also a dynamic trace-comparison tool, first published in 2019 [7] and continuously maintained. Unlike DATA, it reduces traces into a call tree in which each branch represents a CT violation, making comparison faster than traditional trace-alignment algorithms [28]. We set it up following the usage guide.

Pitchfork-angr. Pitchfork-angr [42] (commit `8a3d325`) is a static relational symbolic-execution analyzer for constant-time built on top of the `angr` [71] framework. Like Binsec, it explores secret-dependent divergence in memory access and control flow, but it uses `angr`'s SimuVEX engine and Python tooling rather than Binsec's backend. Like Binsec, it provides bounded soundness. We set it up following the usage guide and hyperparameters following a recent related work [17].