Decompiling for Constant-Time Analysis

SANTIAGO ARRANZ-OLMOS, MPI-SP, Germany
GILLES BARTHE, MPI-SP, Germany and IMDEA Software Institute, Spain
LIONEL BLATTER, MPI-SP, Germany
YOUCEF BOUZID, ENS Paris-Saclay, France
SÖREN VAN DER WALL, TU Braunschweig, Germany
ZHIYUAN ZHANG, MPI-SP, Germany

The constant-time programming discipline is commonly used to protect cryptographic libraries against side-channel attacks. However, it is hard to write constant-time code; moreover, compilers can introduce constant-time violations. Therefore, it is important to ensure that assembly code is constant-time. One approach is to show that source programs are constant-time, and that constant-timeness is preserved by compilation. In this paper, we explore the methodological soundness and scalability of the Decompile-then-Analyze approach, a less conventional alternative that has been suggested in the broader setting of static analysis. Informally, the Decompile-then-Analyze approach uses decompilers a front-end for static analysis tools. As a motivation for our study, we show that current decompilers eliminate CT vulnerabilities before CT analysis, leading to non-CT programs being accepted as constant-time. Independently, we provide *constructed* examples of non-CT, exploitable, programs that are accepted by two popular CT analysis tools; in both cases the culprit are program transformations that are used internally prior to CT analysis and eliminate CT violations. While our examples do not invalidate the general approach of these tools, they emphasize the need for studying the Decompile-then-Analyze approach.

On the methodological side, we define the notion of *CT transparency*. Informally, a program transformation is CT transparent if does not eliminate nor introduce CT violations. We also provide general methods for proving that a transformation is CT transparent, and show that several transformations of interest are transparent. We also sketch an extension of CT transparency to speculative constant-time, which is used by cryptographic software as a protection against Spectre attacks.

On the practical side, we build a CT-transparent version of the popular LLVM-based decompiler RetDec, and combine it with CT-LLVM, an existing CT verification tool for LLVM. We evaluate the resulting tool, called CT-RetDec on a benchmark set of real-world vulnerabilities in binaries, and show that the modifications had significant impact on how well CT-RetDec performs.

CCS Concepts: • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: Decompilation, Static Analysis, Constant-Time, Side Channel

ACM Reference Format:
Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Youcef Bouzid, Sören van der Wall, and Zhiyuan Zhang. 2026. Decompiling for Constant-Time Analysis. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 93 (April 2026), 30 pages. https://doi.org/10.1145/3798201

Authors' Contact Information: Santiago Arranz-Olmos, MPI-SP, Bochum, Germany, santiago.arranz-olmos@mpi-sp.org; Gilles Barthe, MPI-SP, Bochum, Germany and IMDEA Software Institute, Madrid, Spain, gilles.barthe@mpi-sp.org; Lionel Blatter, MPI-SP, Bochum, Germany, lionel.blatter@mpi-sp.org; Youcef Bouzid, ENS Paris-Saclay, Gif-sur-Yvette, France, youcef.bouzid@ens-paris-saclay.fr; Sören van der Wall, TU Braunschweig, Braunschweig, Germany, s.van-der-wall@tu-bs.de; Zhiyuan Zhang, MPI-SP, Bochum, Germany, zhiyuan.zhang@mpi-sp.org.

ACM 2475-1421/2026/4-ART93
https://doi.org/10.1145/3798201

1 Introduction

Decompilers[1] are routinely used in vulnerability analysis to transform binary programs into source or IR programs on which (manual) analysis can be carried out. To maximize their benefits, decompilers aim to produce source or IR programs that are both readable and correctly capture the behavior of their corresponding binary programs. Of course, achieving correctness and readability simultaneously is intrinsically hard. Yet, in spite of the challenge, there has been significant progress towards this goal, through a combination of technical developments [17, 21, 22, 38, 57, 69, 74], and extensive evaluations [7, 24, 30, 32, 46, 49, 68].

More recently, researchers have started to explore the possibility to use decompilation to conduct static analysis for properties such as memory layout and memory safety violations [47, 48, 72]. The idea is simple: take a binary program, use a decompiler to produce an IR or source program, and finally run a static analysis on the source or IR. While less direct, this approach offers complementary benefits over binary-level analysis, in situations where source-level analysis tools are more precise, more scalable, or offer features not supported by their binary-level counterparts.

Problem Statement. In this paper, we consider the question of using decompilation for constant-time analysis—see [11, 35, 42] for an overview of the field. Constant-time (CT) analysis aims to ensure that programs are protected against timing side-channel attacks, in which the attacker recovers secrets by observing the execution of a victim program. The prominent timing side-channel attacks observe the accessed memory locations [18] and the control flow of the victim program [44, 50]. The constant-time policy mitigates these side-channel attacks: It mandates that the program does not perform secret dependent memory accesses and does not branch on secrets.

The main reasons to study decompilation for constant-time analysis are:

(1) The class of attacks mitigated by the constant-time policy, namely timing side-channel attacks, is devastating. In particular, they allow (possibly remote) attackers to recover cryptographic keys very efficiently [18, 44].
(2) The constant-time policy, despite its relative simplicity, is very hard to achieve, even for expert cryptographic developers.
(3) Many popular CT analysis tools[2] operate at source or IR level. These tools can not be used to analyze binaries without a decompiler.
(4) Mainstream compilers introduce violations of the CT policy (CT violations) to binary programs [27, 34, 56, 61], and there is little prospect that the issue will be fixed in the near term. These CT violations are undetectable in the source code.

Therefore, using decompilation for constant-time analysis has the potential to improve a pressing and as yet unresolved security problem. In order to evaluate this potential, our work addresses the following research questions:

RQ1 Are existing decompilers suitable for being used in combination with constant-time analysis?
RQ2 Are there guiding principles to make decompilers suitable to constant-time analysis?
RQ3 Is there a practical decompiler that adheres to these principles and can be used for constant-time analysis of real-world code bases?

We answer **RQ1** negatively by exhibiting a real-world binary program that contains an exploitable CT violation. The program is decompiled by RetDec, a state-of-the-art LLVM decompiler, into a constant-time LLVM IR program that is (rightly) proved secure by LLVM CT analysis tools. Our binary program stems from the reference implementation of ML-KEM, a recently standardized

[1] See https://decompilation.wiki/ for an overview and pointers to the literature.
[2] See https://crocs-muni.github.io/ct-tools/ for a list of tools.

key encapsulation mechanism that is provably secure against quantum adversaries. However, it is vulnerable to the Clangover (CVE-2024-37880) compiler-induced side-channel vulnerability.

We answer **RQ2** positively by introducing the notion of CT transparent transformations. Informally, a transformation is CT transparent if it neither introduces nor eliminates CT violations, generating no false positives or false negatives in the analysis results. We develop rigorous techniques for proving that program transformations are CT transparent by extending CT simulations, originally developed in Barthe et al. [15] to prove constant-time preservation, which is a weaker property than CT transparency. Specifically, we identify a condition on simulations, called $\mathcal{PC}$*-injectivity*, which allows us to reuse existing simulations from Barthe et al. [15] and related works, to prove CT transparency with little additional cost. Using our techniques, we show that many common program transformations are CT transparent; for others, we provide simple examples to explain where and why they fail to be transparent.

We answer **RQ3** positively by developing a toolchain, called CT-RetDec, that combines a modified version of the decompiler RetDec and CT-LLVM, a CT analysis tool for LLVM IR. We then use CT-RetDec to analyze binary code to find CT violations.

As a contribution of independent interest, we observe that CT-Verif [6] and BinSec [27], two popular CT analysis tools, use non-CT transparent converters to transform input programs into another representation on which the CT analysis is performed. In both cases, we can leverage non-transparency of converters to craft *constructed* non-CT examples that are accepted by the tools. While our examples do not invalidate the general approach of these tools for real-world code, we recommend that CT analysis tools rely on transparent converters, and that such converters are clearly exposed in the implementation.

Summary of Contributions. In summary, our main contributions are:

- A demonstration that state-of-the-art decompilers are not transparent and cannot be used for constant-time analysis;
- A formalization of transparency and techniques to prove that a transformation is transparent;
- An analysis of common program transformations, providing for each of them a proof of transparency, or a counterexample to transparency;
- A toolchain, called CT-RetDec, for analyzing binary code against the CT policy, built with transparent decompilation in mind; and
- A study of the transparency of converters employed in CT analysis tools.

Concurrent Approaches to CT Analysis. We refer to the approach of applying CT analysis after decompilation as the Decompile-then-Analyze approach. This approach has been adopted implicitly as an internal component in prior CT analysis tools, such as CacheS [66], and has also been used explicitly as an external front-end, for example by decompiling binaries with Ghidra when analyzing cryptographic protocol implementations [51].

There are two concurrent approaches to deliver CT guarantees for binary code next to the Decompile-then-Analyze approach: Analyze-then-Compile and Binary Analysis. Analyze-then-Compile performs source- or IR-level analysis in order to prove that the program does not violate the CT policy, and then compiles the program into the binary. However, it is well-known that secure source programs can be compiled into binary programs that contain CT violations, leading to a dangerous compiler security gap [31, 61, 67]. Secure compilation aims to address this gap by integrating security considerations into compiler developments [3, 4]. A specific line of work within secure compilation focuses on preserving CT by compilation [13, 15, 16]. However, as stated above, there is little prospect of mainstream compilers to adopt these CT preservation guarantees.

The Binary Analysis approach performs CT analysis on the binary-level, using e.g., BinSec [27]. However, many approaches to CT analysis depend on general-purpose analyses, e.g., alias analysis,

that are easier to perform at IR level. Furthermore, CT analysis may involve complex reasoning that goes beyond automated methods, e.g., to justify declassifying some information, or to reason about leakage in refined models. For such cases, binary-level analysis is not an option.[3]

Paper Structure. Section 2 presents a motivating example and empirical test results, demonstrating that transparency failures in decompilers are a systematic issue. Sections 3 and 4 give a formal definition of CT transparency and a technique for proving transparency. Section 5 investigates the transparency of common transformations. Then, Section 6 introduces CT-RetDec and evaluates its performance in finding CT violations. Section 7 presents two CT analysis tools that fail transparency on constructed input binaries. Finally, Section 8 extends the formalism to speculative constant-time.

Artifact. We present an artifact containing a mechanization of the general version of Theorem 4.5 and the experiments described in the paper in [8].

Extended Version. The extended version of this paper includes four appendices and may be found in https://doi.org/10.48550/arXiv.2501.04183.

2 Motivating Example and Impact

In this section, we investigate the Decompile-then-Analyze approach on a concrete real-world side-channel vulnerability to motivate our study of CT transparency. We discover that decompilation removes the vulnerability from the program, so that subsequent CT analysis (correctly) reports the absence of any vulnerability. We then empirically test five state-of-the-art decompilers on artificial, minimal counterexamples to demonstrate that this issue is not limited to a single decompiler, but happens systematically throughout practical tools.

2.1 Clangover: A Vulnerability Hidden by Decompilation

Clangover is a real-world side-channel vulnerability (CVE-2024-37880), that exists in the binary program of the reference implementation of ML-KEM, the standardized post-quantum key encapsulation mechanism. The vulnerability is in the `poly_frommsg(r, msg)` function that turns a secret message (`msg`) into the coefficients of a polynomial bit by bit. To do so, it loops over `msg`, and for each bit, it sets the corresponding coefficient of the polynomial pointed by `r` to either zero or a constant. The left-hand side of Listing 1 displays the vulnerable excerpt of the binary code of `poly_frommsg` in `x86-64` syntax. It is the `for`-loop that iterates over each byte's bits, extracts the bit, and sets the coefficient for that bit. Line 4 loads the current byte of `msg` from memory and Line 6 extracts the current bit from it. Line 7 branches on the extracted bit in order to write the correct coefficient to memory in Line 10. The conditional branch on the secret bits of `msg` constitutes the CT violation: it is a branching condition that depends on a secret.

To employ the Decompile-the-Analyze approach, we feed the binary program into the off-the-shelf decompiler RetDec, a state-of-the-art decompiler that emits LLVM IR. We then analyze the decompiled program using CT-Verif, a state-of-the-art CT analysis tool for LLVM IR [6]. The corresponding decompiled `for`-loop is shown on the right-hand side of Listing 1, where we manually recovered sensible variable names for readability. Critically, the decompiler has replaced the conditional branch on the secret bits by a conditional move instruction using the `?` operator (Line 7). This removes the CT violation, as conditional move instructions do not leak their conditional [41]. Indeed, CT-Verif correctly identifies that the decompiled program is CT. Hence, using RetDec for the Decompile-then-Analyze approach misses CT vulnerabilities.

[3]Admittedly, our paper focuses on the baseline CT policy and does not deal with declassification nor leakage in refined models, and leaves these extensions for future work.

Listing 1 Decompiling Clangover removes the CT vulnerability.

```
; rsi:msg - rax:msg_offset - r8d:byte
; rcx:j - rdx:coef - rdi:poly_offset
.LBB0_2: ; for(j = 0; j < 8; j++)
    movzx   r8d, byte ptr [rsi + rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .LBB0_4 ; leaks bit of msg
    mov     edx, 1665
.LBB0_4:
    mov     word ptr [rdi + 2*rcx], dx
    inc     rcx
    cmp     rcx, 8
    jne     .LBB0_2
```

```
for (int64_t j = 0; j < 8; j++) {
  byte = *(char *)(msg_offset + msg);

  coef = (1 << (int32_t)j % 32 & (int32_t)byte) == 0 ?
            0 : 1665;       // no leak

  *(int16_t *)(2 * j + poly_offset) = coef;

}
```

Table 1. Analysis of five decompilers. A ↯ (resp. -) means the decompiler removes (resp. keeps) the CT violation.

Example	Angr	BinaryNinja	Ghidra	Hex-Rays	RetDec	Root cause
Listing 1	↯	-	-	-	↯	If Conversion
Listing 2	↯	↯	-	↯	↯	Branch Coalescing
Listing 3	-	↯	↯	↯	↯	Empty Branch Coalescing
Listing 4	↯	↯	↯	↯	↯	Dead Load Elimination
Listing 5	-	-	-	-	↯	Dead Store Elimination

2.2 Transparency Failures in Decompilers

The previous section highlights the risk of the Decompile-then-Analyze approach on a singular binary program and decompiler. This naturally raises the question of whether it is an isolated incident, or whether this is a systematical issue in decompilers. We address this question by empirically testing state-of-the-art decompilers on carefully crafted non-CT binary code snippets. They are minimal examples that contain CT violations. The CT violations leak a secret through either a dead load/store instruction, or a spurious or simple branch. We defer the presentation of the examples and the offending transformations to Section 5.3. Our test spans five decompilers: Angr 9.2.127 [60], BinaryNinja 4.1.5747 [1], Ghidra 11.2 [52], Hex-Rays 8.4.0.240320 [39] and RetDec 5.0 [45] using the Decompiler Explorer [2]. In Table 2, we report if a decompiler removes or keeps the CT violation. The result is that each of the five decompilers removes CT violations. Therefore, the issue is indeed systematic throughout practical decompilers.

3 Transparency

This section reviews the constant-time property using an abstract model of computation and introduces the notion of transparent transformations.

We keep the programming language abstract and say that $\mathcal{L}$ is a set of *programs*. The semantics of a program is given by transitions on *states* $\mathcal{S}$ that capture, for instance, the values of registers and memory and the current program point, at a point in time. These transitions constitute a relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$, where we write $s \rightarrow s'$ to indicate that a state s transitions to state s'. The language also specifies which states are *final*, i.e., states where the program has terminated. We assume that the semantics is deterministic, i.e., if $s \rightarrow s'$ and $s \rightarrow s''$ then $s' = s''$, and safe, i.e., for every s, either s is final, or there exists s' such that $s \rightarrow s'$.

Leakage. We extend this model to capture side-channel leakage by instrumenting transitions with *observations* (denoted O): the transition $s \xrightarrow{o} s'$ indicates that going from s to s' emits the

$$\textsc{refl}\ \frac{}{s \xrightarrow{\epsilon}{}^* s} \qquad \textsc{trans}\ \frac{s \xrightarrow{o} s' \qquad s' \xrightarrow{\boldsymbol{o}}{}^* s''}{s \xrightarrow{o \cdot \boldsymbol{o}}{}^* s''}$$

Fig. 1. Multi-step execution semantics.

observation $o \in \mathcal{O}$. Observations represent what is leaked by the transition, i.e., what an attacker can learn. For example, in the constant-time leakage model, a transition that performs a memory load emits an observation that contains the address of the load, and a conditional branch emits an observation containing its condition. These values are leaked because an attacker who observes the data and instruction caches can compute them, respectively.

In this work, we assume that observations reveal the control flow of a program [50]. To do so, we assume that every state s has a *program point* (we write $\mathcal{PC}$ for the set of program points), denoted $\mathrm{pc}(s)$. For example, the program point in an assembly language is the program counter, and in a structured language it is the remaining code to be executed. Formally, the assumption that observations reveal the program point means: for every two states at the same program point $\mathrm{pc}(s_1) = \mathrm{pc}(s_2)$, that step with the same observation $s_1 \xrightarrow{o} s_1'$ and $s_2 \xrightarrow{o} s_2'$, we have that the resulting states are also at the same program point, i.e., $\mathrm{pc}(s_1') = \mathrm{pc}(s_2')$.

Program Behavior. We define the behavior of programs in terms of executions, which comprise multiple steps. Figure 1 defines executions as the reflexive, transitive closure $s \xrightarrow{\boldsymbol{o}}{}^* s'$ of $\rightarrow$. We accumulate the observations leaked during the execution in a list, denoted with a bold $\boldsymbol{o}$. The length of $\boldsymbol{o}$, denoted $|\boldsymbol{o}|$, matches the number of steps in the execution.

A program $P \in \mathcal{L}$ starts its execution with a list of *input values*, taken from an input domain $\mathcal{I}$. Inputs capture the interaction of the program with its environment, e.g., the arguments given to the entry point of a program. The inputs fully determine the initial state of the program, thus we write $P(i) \in \mathcal{S}$ for the initial state of P on inputs $i \in \mathcal{I}$. The behavior of a program are the observations along all its executions, as follows.

Definition 3.1 (Program Behavior). The behavior $Beh(P, i)$ of program P on input i is the set of leakage traces starting from $P(i)$, i.e., $Beh(P, i) \triangleq \{\boldsymbol{o} \mid P(i) \xrightarrow{\boldsymbol{o}}{}^* s\}$.

Constant-Time. As usual with noninterference definitions, our security property is based on an *indistinguishability* relation $\phi \subseteq \mathcal{I} \times \mathcal{I}$ on inputs. Indistinguishability expresses what part of the input is public. For example, if the inputs to a program are a secret message m and its public length n, it is appropriate to define $(m, n)\ \phi\ (m', n') \triangleq n = n'$, as the secret message is not known to an attacker, while the length of the message is known. The intuition is that an adversary cannot distinguish the two inputs (m, n) and (m', n') if the public part coincides, i.e., the lengths n and n' are the same.

A program P is *constant-time* w.r.t. an indistinguishability relation ϕ (denoted P is ϕ-CT) if the observations generated by executions starting from indistinguishable inputs produce the same observations. We use the term *ϕ-CT violation* to refer to a difference in the observations generated by a program.

Definition 3.2 (ϕ-CT). P is ϕ-CT if for all pairs of inputs $i_1\ \phi\ i_2$ we have $Beh(P, i_1) = Beh(P, i_2)$.

Transparency. We now turn to the interactions between program transformation and constant-time. Consider a transformation $(\!|\cdot|\!) : \mathcal{L}_s \rightarrow \mathcal{L}_t$ that maps programs in an *input* language, denoted $\mathcal{L}_s$, to programs in an *output* language, denoted $\mathcal{L}_t$ (throughout the paper, we use these subscripts

to refer to input and output languages). We want to reason about whether $(\!|\cdot|\!)$ introduces or removes ϕ-CT violations. We consider three relevant properties, as follows.

Definition 3.3 (Reflection, Preservation, and Transparency). We say that a program transformation $(\!|\cdot|\!) : \mathcal{L}_s \to \mathcal{L}_t$ between an input language $\mathcal{L}_s$ and an output language $\mathcal{L}_t$

- Reflects CT if for each P and ϕ, $(\!|P|\!)$ is ϕ-CT implies P is ϕ-CT;
- Preserves CT if for each P and ϕ, P is ϕ-CT implies $(\!|P|\!)$ is ϕ-CT; and
- Is CT transparent if it both reflects and preserves CT.

Indeed, if $(\!|\cdot|\!)$ reflects CT, it does not remove a ϕ-CT violation for any indistinguishability relation ϕ and input program P. Similarly, if $(\!|\cdot|\!)$ preserves CT, it does not introduce any ϕ-CT violations.

4 Proof Techniques

In this section, we present a method to prove the CT transparency of a program transformation, drawing on existing work in the area of secure compilation that targets CT preservation, for instance [13, 15, 16].

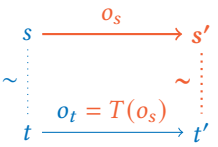
We aim to establish a *simulation* between the input program and output program of the transformation. A simulation links states of the input and the output programs so that the transitions of the output program can be mimicked by the input program. In order to preserve CT, this approach additionally relates input and output observations: output observations must be expressible as a function of input observations, which is called the observation transformer. To extend this approach to CT transparency, our simulations pose an additional requirement on the observation transformers.

We start our presentation with a simplified version of our simulations to build some intuition, and present the general version afterward.

Lock-Step Simulations for CT Transparency. The basic form of simulations are *lock-step* simulations, where every step of the output program is mimicked by precisely one step of the input program.

Definition 4.1 (Lock-Step Simulation Diagram). A relation $\sim \subseteq \mathcal{S}_s \times \mathcal{S}_t$ satisfies a lock-step simulation diagram w.r.t. an observation transformer $T : O_s \to O_t$ if for every pair of related states $s \sim t$ in which the output program steps $t \xrightarrow{o_t} t'$, then the input program steps $s \xrightarrow{o_s} s'$ such that $s' \sim t'$ and $o_t = T(o_s)$.

In the diagram, the relation on the left and the step at the bottom (in thin blue) are premises, and the relation on the right and the step at the top (in **bold red**) are conclusions.



Intuitively, the observation transformer T explains the observations of the output program as a function of the observations of the input program. This guarantees preservation because if the input program is ϕ-CT, indistinguishable inputs give equal observations $o_s = o'_s$, and, thus, also the output program produces the same observations $T(o_s) = T(o'_s)$.

CT reflection, however, aims for the dual of preservation: we want that if the output program is ϕ-CT, then the input program should be ϕ-CT as well (recall Definition 3.3). Rather than crafting a new kind of simulation, we identify an additional requirement on T that guarantees CT reflection: injectivity. If the observation transformer is injective, we can flip the argument around: when the output program's observations are equal, $o_t = T(o_s) = T(o'_s) = o'_t$, injectivity makes the input program's observations equal as well, $o_s = o'_s$. The following theorem identifies the sufficient

conditions for using a lock-step simulation to prove that a transformation also reflects CT, i.e., that it is CT transparent.

Theorem 4.2 (Soundness of Lock-Step Simulations). *A program transformation* $(\!|\cdot|\!) : \mathcal{L}_s \to \mathcal{L}_t$ *is CT transparent if for every input program* P *there exist* $\sim$ *and* T *such that*

(i) $\sim$ *satisfies a lock-step simulation diagram w.r.t.* T*;*
(ii) *Initial states are related:* $P(i) \sim (\!|P|\!)(i)$ *for every* i*; and*
(iii) T *is injective.*

Note that this theorem ensures transparency with respect to *every* indistinguishability relation ϕ even though the simulation relation $\sim$ and the observation transformer T do not depend on ϕ.

The critical requirement for CT reflection in Theorem 4.2 is that the observation transformer T is injective, that is, that equal output observations imply equal input observations. Since prior work has shown that lock-step simulations guarantee preservation, and our condition guarantees reflection, we obtain transparency.

Lock-step simulations are too constraining for many standard transformations. We introduce a relaxed version of simulations that allows us to prove reflection for all transformations considered in the remaining paper.

Relaxed Simulations for CT Transparency. In contrast to lock-step simulations, relaxed simulations allow the input and output programs to perform different number of steps. To this end, when the output program steps, we allow the input program to make any positive number of steps. We introduce a function $\mathsf{ns} : \mathcal{PC} \to \mathbb{N}_{>0}$ that determines the *number of steps* the input program performs to "catch up" with the output program, i.e., such that the states are related by $\sim$ again. Requiring that the number of steps is nonzero simplifies our definitions, and is easily satisfied in all our use cases, since decompiler transformations rarely introduce instructions in the output program—their aim is to simplify the program. Consequently, some transformations remove instructions from the input program, which means that the output program reaches a final state before the input program does. In these cases, we must justify the leakage of the input program without an output program step: we introduce a *suffix* function $\mathsf{sf} : \mathcal{PC} \to O_s^*$, which determines the remaining observations of the input program when it reaches a program point where the output program has terminated.

In addition to these relaxations, the observation transformer T becomes a partial function that may additionally inspect program point of the source state to transform observations, i.e., it takes an extra argument. These extensions allow relaxed simulations to accommodate all transformations of this work. We now recast Definition 4.1 and Theorem 4.2.

Definition 4.3 (Simulation Diagram). A relation $\sim \subseteq \mathcal{S}_s \times \mathcal{S}_t$ satisfies a simulation diagram w.r.t. a partial observation transformer $T : \mathcal{PC} \times O_s^* \rightharpoonup O_t$, a number-of-steps function ns, and a suffix function sf, if for all related states $s \sim t$, the input program can execute $s \xrightarrow{\boldsymbol{o}_s}{}^* s'$ such that

(i) If the output program steps $t \xrightarrow{o_t} t'$, then the input program takes $|\boldsymbol{o}_s| = \mathsf{ns}(\mathsf{pc}(s))$ steps from $s \xrightarrow{\boldsymbol{o}_s}{}^* s'$, so that $o_t = T(\mathsf{pc}(s), \boldsymbol{o}_s)$ and $s' \sim t'$.

(ii) If t is final instead, then the source observations $\boldsymbol{o}_s$ are $\mathsf{sf}(\mathsf{pc}(s))$, s' is final, and $s' \sim t$.

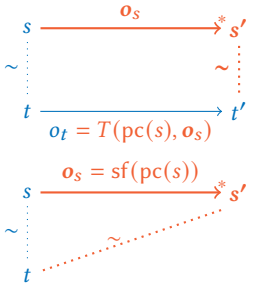


Table 2. Summary of decompilation passes in this section. A ✓ means that we prove the pass transparent in Section 5.2, a ↯ that we present a counterexample to CT reflection in Section 5.3, and • means the pass is CT preserving, but we omit the proof.

Pass	Reflection	Preservation
Branch Coalescing	↯	•
If Conversion	↯	•
Memory Access Elimination	↯	•
Constant Folding	✓	✓
Untiling	✓	✓
Dead Branch Elimination	✓	✓
Dead Assignment Elimination	✓	✓
Unspilling	✓	✓
Structural Analysis	✓	✓
Loop Rotation	✓	✓

We must overcome a final challenge to achieve transparency in this relaxed setting: we redefine the injectivity requirement (iii) from Theorem 4.2, since now the observation transformer T inspects program points and is partial. First we require injectivity on T *per program point*. Second, we require injectivity only where T is defined. We call this new condition $\mathcal{PC}$*-injectivity*.

Definition 4.4 ($\mathcal{PC}$-Injectivity). A partial observation transformer $T : \mathcal{PC} \times \mathcal{O}_s^* \to \mathcal{O}_t$ is $\mathcal{PC}$-injective, when for each $\mathsf{pc} \in \mathcal{PC}$,

$$T(\mathsf{pc}, \boldsymbol{o}_1) = T(\mathsf{pc}, \boldsymbol{o}_2) \neq \bot \implies \boldsymbol{o}_1 = \boldsymbol{o}_2.$$

Theorem 4.5 (Soundness of Simulations). *A program transformation $(\!|\cdot|\!) : \mathcal{L}_s \to \mathcal{L}_t$ is CT transparent if for every input program P there exist a relation $\sim$, a number-of-steps function* ns*, an observation transformer T, and a suffix function* sf*, such that*

(i) *$\sim$ satisfies a simulation diagram w.r.t. T,* ns*, and* sf*;*
(ii) *Initial states are related: $P(i) \sim (\!|P|\!)(i)$ for every i; and*
(iii) *T is $\mathcal{PC}$-injective.*

For a detailed proof of this theorem, we refer the reader to our Rocq development in the artifact. In this mechanized proof, we split Theorem 4.5 into two parts: Theorem `th2_preserves` states that simulations ensure preservation, and Theorem `th2_reflect` states that simulations guarantee reflection. In our formal development, we further generalized some of the definitions, which we simplified in the text for better presentation.

5 Transparent and Nontransparent Transformations

This section considers the CT transparency of ten common transformations. We prove that seven of them are transparent, provide counterexamples for the rest, and summarize the results in Table 2.

5.1 Language and Leakage Model

To prove CT transparency of the program transformations, we instantiate our framework from Section 3 with a core language of register assignments, memory operations, and loops. The syntax of the language is as follows:

$$e ::= z \mid b \mid x \mid \oplus(e, \cdots, e), \qquad a ::= x = e \mid x = [e] \mid [e] = x,$$
$$c ::= \texttt{skip} \mid c\,;c \mid a \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c,$$

$$\textsc{Assign}\ \frac{\rho' = \rho[x \leftarrow [\![e]\!]_\rho]}{\langle x = e, \rho, \mu\rangle \xrightarrow{\bullet} \langle \texttt{skip}, \rho', \mu\rangle} \qquad \textsc{Load}\ \frac{n = [\![e]\!]_\rho \qquad \rho' = \rho[x \leftarrow \mu(n)]}{\langle x = [e], \rho, \mu\rangle \xrightarrow{\text{adr } n} \langle \texttt{skip}, \rho', \mu\rangle} \qquad \textsc{Store}\ \frac{n = [\![e]\!]_\rho \qquad \mu' = \mu[n \leftarrow \rho(x)]}{\langle [e] = x, \rho, \mu\rangle \xrightarrow{\text{adr } n} \langle \texttt{skip}, \rho, \mu'\rangle}$$

$$\textsc{Cond}\ \frac{[\![e]\!]_\rho = b}{\langle \texttt{if } e \texttt{ then } c_{\text{tt}} \texttt{ else } c_{\text{ff}}, \rho, \mu\rangle \xrightarrow{\text{br } b} \langle c_b, \rho, \mu\rangle} \qquad \textsc{While}\ \frac{[\![e]\!]_\rho = b \qquad c_{\text{tt}} = c; \texttt{while } e \texttt{ do } c \qquad c_{\text{ff}} = \texttt{skip}}{\langle \texttt{while } e \texttt{ do } c, \rho, \mu\rangle \xrightarrow{\text{br } b} \langle c_b, \rho, \mu\rangle}$$

$$\textsc{Seq}\ \frac{\langle c, \rho, \mu\rangle \xrightarrow{o} \langle c', \rho', \mu'\rangle}{\langle c\,; c'', \rho, \mu\rangle \xrightarrow{o} \langle c'\,; c'', \rho', \mu'\rangle}$$

Fig. 2. Single-step execution semantics.

where z is an integer, b is a boolean, x is a register variable, and $\oplus$ an operation (such as negation $\neg$ or addition +). Expressions e are built from constants and register variables but may not invoke memory accesses, this means that they have no side effects and no leakage. Atomic commands a are assignments, loads, and stores. Commands c are the standard while language constructs. We make the usual assumptions that the empty program `skip` is neutral for sequentialization `skip`$\,; c = c\,;$ `skip` $= c$, and that sequencing is associative $(c\,; c')\,; c'' = c\,;(c'\,; c'')$.

The semantics operates on states $\langle c, \rho, \mu\rangle$ consisting of the command c remaining to be executed, a register map ρ that associates each register variable with a value, and a memory μ that associates each address with a value. The program point of a state is its code, i.e., $\text{pc}(\langle c, \rho, \mu\rangle) \triangleq c$, and a state is final if its code is `skip`. Figure 2 presents the small-step semantics $s \xrightarrow{o} s'$ of commands. We choose the standard constant-time leakage model [15] for our semantics, i.e., memory operations leak their address (adr n in Load, Store) and branch instructions leak their condition (br b in Cond, While). The semantics and leakage model satisfy the requirements of Section 3 (it is deterministic, and programs are safe).

5.2 Constant-Time Transparent Transformations

We now examine the seven transparent transformations in Table 2 and provide proof sketches for their transparency with the framework from Section 4. We will keep the transformations as abstract as possible and focus on their transparency rather than the details of their functionality. When convenient, we split passes into an analysis, which annotates the program, followed by a program transformation, which expects these annotations in the input program. We present detailed definitions of the transformations and proofs of transparency in Appendix A.

Expression Substitution. This transformation replaces expressions with other expressions that produce the same values (it generalizes Constant Folding and Untiling). For example, it may replace $3 * x - x$ with $2 * x$. It may, as well, rely on the program's structure and replace the instruction sequence $x = y * 8\,; x = [z + x]$ with $x = y * 8\,; x = [z + (y * 8)]$. In general, Expression Substitution is preceded by an analysis that identifies and annotates all expressions that is to be replaced. We write an annotated expression as $\{e_s \rhd e_t\}e$, which means that every expression e_s occuring in e are to be replaced by e_t. Whichever analysis is used, as long as it guarantees the following criterion, that states that the expressions e_s and e_t indeed yield the same values, we can prove the actual transformation transparent.

Proposition 5.1. *For all inputs i and executions $P(i) \xrightarrow{o}{}^{*} \langle c, \rho, \mu\rangle$, if the next instruction of c contains an annotated expression $\{e_s \rhd e_t\}e$, then $[\![e_s]\!]_\rho = [\![e_t]\!]_\rho$.*

A few typical passes in static analysis tools and decompilers are subsumed by this definition. Among them are Constant Folding and Untiling. Constant Folding replaces expressions without variables by an appropriate constant (our first example). Untiling is a decompilation pass that typically sequences of instructions into a single, more complex instruction. This is desirable when compilers split complex expressions into multiple instructions in order to compute the expression on hardware, but the complex expression is more readable. Our second example from above is a case of untiling: the first instruction $x = y * 8$ computes the address offset for the second instruction $x = [z + x]$, which are merged into a single instruction $x = [z + (y * 8)]$. Notice we define Expression Substitution to not remove the first instruction $x = y * 8$. This step is left to Dead Assignment Elmination, for which we prove transparency in a seperate transformation.

Let us turn to proving the transparency of Expression Substitution. The transformation maintains the structure of the program and the number of statements. Therefore, we define a lock-step simulation and apply Theorem 4.2. This involves defining a relation $\sim$ and a function T such that

(i) $\sim$ satisfies a lock-step simulation diagram w.r.t. T;
(ii) $P(i) \sim (\!| P |\!)(i)$ for every i; and
(iii) T is injective.

Theorem 5.2. *Expression Substitution is CT transparent.*

Proof Sketch. We define the simulation relation $\sim$ such that $\langle c, \rho, \mu\rangle$ is related only to $\langle (\!| c |\!), \rho, \mu\rangle$. That is, the code in the output program's state is the transformed input program's code at that state, and the register map as well as the memory coincides. The leakage transformer is the identity: $T(o_s) = o_s$. This satisfies the requirements of Theorem 4.2 as follows:

(i) Since the evaluated expressions produce the same value (Proposition 5.1), every step in the output program matches exactly one step in the input program, and they produce the same register map and memory. Since the register map and memory are identical, the input and output observations are equal. For instance, the instructions $x = [z + y * 8]$ and $x = [z + x]$ from the above example produce the same observation, namely adr $[\![z + y * 8]\!]$.
(ii) Initial states are trivially related.
(iii) T is injective because it is the identity. □

Dead Branch Elimination. This transformation eliminates conditional branches that are never taken. That is, it transforms conditionals `if` b `then` c_{tt} `else` c_{ff}, where $b \in \{\mathsf{tt}, \mathsf{ff}\}$ is a constant, into the corresponding branch c_b. The transformation modifies no other instructions. Intuitively, this pass is transparent because a dead branch is never executed, and, therefore, removing it does not affect leakage. Also, removing the leak that stems from the branching instruction itself is not an issue, because the constant b is secret-independent.

Theorem 5.3. *Dead Branch Elimination is CT transparent.*

Proof Sketch. Since Dead Branch Elimination removes code, the input and output programs perform a different number of steps. This is the setting to use our relaxed version of simulations and apply Theorem 4.5. It requires us to provide $\sim$ and T as before, but additionally requires the number of steps function ns and the suffix transformer sf. We define $\sim$ to only hold on $\langle c, \rho, \mu\rangle \sim \langle (\!| c |\!), \rho, \mu\rangle$ as before, and we provide the remaining instantiations in Figure 3.

Our definitions satisfy the requirements of Theorem 4.5:

(i) We prove that $\sim$ satisfies Definition 4.3 w.r.t. T, ns, and sf, so let $s \sim t$ be given. If t is not a final state, s needs to perform zero or more steps corresponding to the constant branches before performing the same step as t. This number of steps is provided by $\mathsf{ns}(\mathsf{pc}(s))$. Consequently, the input program's observations are of the form $\mathsf{br}\ b_1 \cdot \ldots \cdot \mathsf{br}\ b_n \cdot o_t$, where b_i is the i-th

$$T(c, \boldsymbol{o}) \triangleq \begin{cases} T(c_b\,;c', \boldsymbol{o}') & \text{if } c \text{ is } \texttt{if } b \texttt{ then } c_{\texttt{tt}} \texttt{ else } c_{\texttt{ff}}\,;c' \text{and } \boldsymbol{o} = \text{br } b \cdot \boldsymbol{o}', \\ \boldsymbol{o} & \text{if } c \text{ is not } \texttt{if } b \texttt{ then } c_{\texttt{tt}} \texttt{ else } c_{\texttt{ff}}\,;c' \text{ and } |\boldsymbol{o}| = 1, \\ \bot & \text{otherwise,} \end{cases}$$

$$\mathsf{ns}(c) \triangleq \begin{cases} \mathsf{ns}(c_b;c') + 1 & \text{if } c \text{ is } \texttt{if } b \texttt{ then } c_{\texttt{tt}} \texttt{ else } c_{\texttt{ff}}\,;c', \\ 1 & \text{otherwise,} \end{cases}$$

$$\mathsf{sf}(c) \triangleq \begin{cases} \text{br } b \cdot \mathsf{sf}(c_b;c') & \text{if } c \text{ is } \texttt{if } b \texttt{ then } c_{\texttt{tt}} \texttt{ else } c_{\texttt{ff}}\,;c', \\ \epsilon & \text{otherwise.} \end{cases}$$

Fig. 3. Relaxed simulation instantiations of T, ns, and sf for Dead Branch Elimination.

$$T(c, \boldsymbol{o}) \triangleq \begin{cases} T(c', \boldsymbol{o}') & \text{if } c \text{ is } x = e\{\mathsf{D}\}\,;c' \text{ with } x \in \mathsf{D} \text{ and } \boldsymbol{o} = \bullet \cdot \boldsymbol{o}', \\ \boldsymbol{o} & \text{if } c \text{ is not } x = e\{\mathsf{D}\}\,;c' \text{ with } x \in \mathsf{D} \text{ and } |\boldsymbol{o}| = 1, \\ \bot & \text{otherwise,} \end{cases}$$

$$\mathsf{ns}(c) \triangleq \begin{cases} \mathsf{ns}(c') + 1 & \text{if } c \text{ is } x = e\{\mathsf{D}\}\,;c' \text{ and } x \in \mathsf{D}, \\ 1 & \text{otherwise,} \end{cases}$$

$$\mathsf{sf}(c) \triangleq \begin{cases} \bullet \cdot \mathsf{sf}(c') & \text{if } c \text{ is } x = e\,;c', \\ \epsilon & \text{otherwise.} \end{cases}$$

Fig. 4. Relaxed simulation instantiations of T, ns, and sf for Dead Assignment Elimination.

dead branch of s. The observation transformer T deletes the branching observations. If t is final, there might remain conditional branches to be executed in s. Therefore, sf provides br b_i observations corresponding to the branches at s.

(ii) As with Expression Substitution, initial states are trivially related by the definition of $\sim$.

(iii) To show $T(c, \boldsymbol{o}_1) = T(c, \boldsymbol{o}_2) \neq \bot \implies \boldsymbol{o}_1 = \boldsymbol{o}_2$, we proceed by induction on the length of $\boldsymbol{o}_1$. There are only two significant cases, corresponding to whether the code c starts with a dead branch. If it does not, the transformer is the identity which makes $\boldsymbol{o}_1 = \boldsymbol{o}_2$ hold trivially. Otherwise, the first observations of both $\boldsymbol{o}_1$ and $\boldsymbol{o}_2$ must be br b, and the remaining observations match by inductive hypothesis. □

Dead Assignment Elimination. This transformation removes assignments to dead variables, i.e., when the assignments have no impact on the execution of the program. For example, it replaces $x = e_1\,;x = [e_2]$ with $x = [e_2]$ because the first assignment is never used. This transformation removes only assignments, *but not loads or stores*. Indeed, removing memory accesses would break CT reflection, as their addresses may leak (see Dead Load Elimination in Section 5.3). We assume that a previous analysis annotated assignment instructions with a set D of the dead variables at that point (we omit the correctness guarantee). The transformation then removes all assignments $x = e\{\mathsf{D}\}$ where $x \in \mathsf{D}$, i.e., the assigned variable is marked dead.

Theorem 5.4. *Dead Assignment Elimination is CT transparent.*

Proof Sketch. The proof of reflection for this pass is similar to Dead Branch Elimination, i.e., we apply Theorem 4.5. We define the simulation relation $\langle c, \rho, \mu\rangle \sim \langle c', \rho', \mu'\rangle$ when $c' = (\!|\, c \,|\!)$, ρ and ρ'

coincide everywhere except on dead variables, and $\mu = \mu'$. We define the remaining instantiations in Figure 4. The proof of the requirements of Theorem 4.5 is analogous to Dead Branch Elimination. □

Unspilling. This transformation removes spill-unspill pairs. Compilers emit spill instructions to temporarily store a value to the stack before reloading it later with an unspill. This transformation avoids using the memory by moving such values into fresh variables instead. It allows us to translate an assembly-like language with a limited set of registers to a high-level language with unlimited abstract variables, potentially improving the accuracy of static analyses. A spill-unspill pair consists of a store and a load instructions with the same constant offset from the stack pointer. We denote the stack pointer as sp. For instance, unspilling transforms $[\mathrm{sp} + 4] = x\,; c\,; x = [\mathrm{sp} + 4]$ into $y = x\,; c\,; x = y$, where c does not touch the memory location $\mathrm{sp} + 4$ and y is a fresh temporary variable. Previously, we mentioned that removing memory accesses does not reflect CT. Fortunately, this pass is CT transparent because the addresses leaked by spills and unspills are constant offsets from the stack pointer, which in turn is constant throughout execution—recall that we do not consider function calls in our language.

Theorem 5.5. *Unspilling is CT transparent.*

Proof Sketch. Even though this pass maintains the control flow structure of the program, we cannot use Theorem 4.2 since we need the relaxed version of the observation transformer: we need to inspect the program point in order to transform the observation.

We define $\langle c, \rho, \mu\rangle \sim \langle c', \rho', \mu'\rangle$ to hold when the output program's code is the transformed input program's code, $c' = (\!|\, c \,|\!)$; the variable maps ρ and ρ' coincide except on the fresh temporary variables introduced by the transformation; the memories μ and μ' coincide except on spilled stack offsets; and each spilled offset n holds the value of its fresh variable y, i.e., $\mu(\mathrm{sp} + n) = \rho'(y)$.

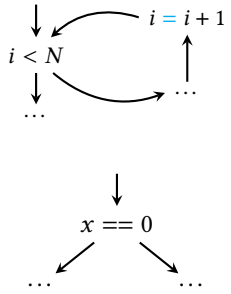
Since each output program step is simulated by exactly one input program step, we define $\mathsf{ns}(s) \triangleq 1$ and $\mathsf{sf}(c) \triangleq \epsilon$. Finally, the observation transformer produces $\bullet$ instead of $\mathrm{adr}\ (\mathrm{sp} + n)$ observations.

$$T(c, \boldsymbol{o}) \triangleq \begin{cases} \bullet & \text{if } c \text{ is } x = [\mathrm{sp} + n]\,; c' \text{ or } [\mathrm{sp} + n] = x\,; c' \text{ and } \boldsymbol{o} = \mathrm{adr}\ (\mathrm{sp} + n), \\ \boldsymbol{o} & \text{if } c \text{ is not } x = [\mathrm{sp} + n]\,; c' \text{ or } [\mathrm{sp} + n] = x\,; c' \text{ and } |\boldsymbol{o}| = 1, \\ \bot & \text{otherwise.} \end{cases}$$

The first two requirements hold straightforwardly. Injectivity on instruction spills and unspills instructions holds since the program point is the same, and otherwise trivially since the transformer returns its argument. □

Structural Analysis. This transformation is the core transformation in binary lifters, which have the goal to take a binary input program and output a structured program. It takes an input program in control flow graph syntax and identifies patterns that correspond to structures such as conditionals and while loops. Then, it replaces the structures found and outputs a program in structured syntax. Structural Analysis sometimes fails when the input CFG program contains patterns that originated from complex control flow structures such as break statements. In this work, we focus on the basic patterns of loops and conditional as branches displayed in Figure 5a. More involved analyses, like single-exit-single-successor analysis [33], can accommodate more complex control flow, based on ideas such as tail regions and iterative analysis [58].

Structural Analysis is an example, where $(\!|\cdot|\!) : \mathcal{L}_s \to \mathcal{L}_t$ transforms the syntax of the programs, i.e., $\mathcal{L}_s \neq \mathcal{L}_t$. For the output program's syntax $\mathcal{L}_t$, we use the structured programs from Section 3. The input language $\mathcal{L}_s$ are control flow graphs that we sketch briefly.



(a)

$$\textsc{Assign}\quad \frac{\ell : x = e \rightarrow \ell' \qquad \rho' = \rho[x \leftarrow [\![e]\!]_\rho]}{\langle \ell, \rho, \mu \rangle \xrightarrow{\bullet} \langle \ell', \rho', \mu \rangle}$$

$$\textsc{Load}\quad \frac{\ell : x = [e] \rightarrow \ell' \qquad a = [\![e]\!]_\rho \qquad \rho' = \rho[x \leftarrow \mu(a)]}{\langle \ell, \rho, \mu \rangle \xrightarrow{\text{adr } a} \langle \ell', \rho', \mu \rangle}$$

$$\textsc{Store}\quad \frac{\ell : [e] = x \rightarrow \ell' \qquad a = [\![e]\!]_\rho \qquad \mu' = \mu[a \leftarrow \rho(x)]}{\langle \ell, \rho, \mu \rangle \xrightarrow{\text{adr } a} \langle \ell', \rho, \mu' \rangle}$$

$$\textsc{Cond}\quad \frac{\ell : e \rightarrow \ell_{\mathrm{tt}}, \ell_{\mathrm{ff}} \qquad b = [\![e]\!]_\rho}{\langle \ell, \rho, \mu \rangle \xrightarrow{\text{br } b} \langle \ell_b, \rho, \mu \rangle}$$

(b)

Fig. 5. (a) Example CFG patterns. The top pattern corresponds to a while loop and the bottom one to a conditional. (b) Semantics of the CFG language.

CFG Language. Our input language consists of assembly-like CFG programs. A CFG program is a set G of labelled nodes. There are two types of nodes: instruction nodes are triples (ℓ, a, ℓ') where ℓ is the label of the node, a is an atomic command (i.e., it is an assignment $x = e$, a load $x = [e]$, or a store $[e] = x$), and ℓ' is the label of the successor node. Branching nodes $(\ell, e, \ell_{\mathrm{tt}}, \ell_{\mathrm{ff}})$ instead contain a branching condition e, and two successors. Each program has a distinguished initial and final label. When G is clear from context, we write $\ell : i \rightarrow \ell'$ when the program contains the instruction node $(\ell, i, \ell') \in G$, and $\ell : e \rightarrow \ell_{\mathrm{tt}}, \ell_{\mathrm{ff}}$ when it contains the branching node $(\ell, e, \ell_{\mathrm{tt}}, \ell_{\mathrm{ff}}) \in G$.

The semantics of this language operates on states $\langle \ell, \rho, \mu \rangle$ consisting of the label ℓ which is the current program point, $\mathrm{pc}(\langle \ell, \rho, \mu \rangle) = \ell$, a register map ρ that associates register variables with values, and a memory μ that associates addresses to values. We provide its leakage semantics $s \xrightarrow{o} s'$ in Figure 5b. It is straightforward to see that the CFG semantics satisfies the constraints from our framework in Section 3.

Theorem 5.6. *Structural Analysis is CT transparent.*

Proof Sketch. We write $\mathit{struct}_G(\ell)$ for the structured code that Structural Analysis identified to be the matching structured program for ℓ in the CFG program G. As mentioned above, the precise definition of $\mathit{struct}_G(\ell)$ can be found in Appendix A.4.

In order to prove transparency we define a lock-step simulation and apply Theorem 4.2. For a input CFG G, the simulation relation $\sim$ is equality on register and memory contents, while the label of the input program is replaced by its structured program, $\langle \ell, \rho, \mu \rangle \sim \langle \mathit{struct}_G(\ell), \rho, \mu \rangle$. We choose the identity as the observation transformer.

Requirement (i) from Theorem 4.2 follows from the fact that struct_G ensures the next instruction to be executed in ℓ always coincides with $\mathit{struct}_G(\ell)$. That fact gives rise to the following proposition that immediately implies Requirement (i) from Theorem 4.2.

Proposition 5.7. *For any pair of related states $s \sim t$, the input state steps $s \xrightarrow{o} s'$ if and only if the output state steps $t \xrightarrow{o} t'$, and, furthermore, $s' \sim t'$.*

The other two requirements are satisfied as well: initial states that share the same inputs are related trivially, because the initial label ℓ_{init} of G maps to the initial program code of $(\!| G |\!)$ and the contents of registers and memory coincide; and T is injective because it is the identity. □

Loop Rotation. This transformation is used to move the entry point of a loop to a desired location in the loop body. Figure 6 illustrates the transformation: the left-hand side presents the input

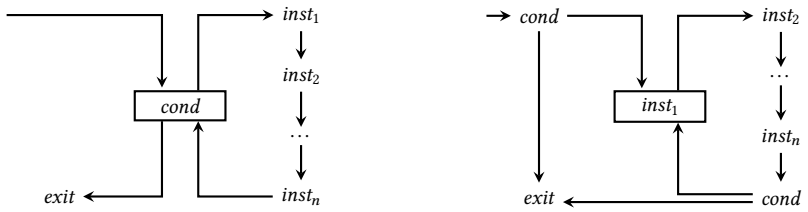


Fig. 6. Loop Rotation duplicates the loop entry and then moves the entry point of the loop by one instruction.

program, where the boxed instruction is the loop entry, and the right-hand side is the output program. The transformation duplicates the loop entry, *cond*, and makes its successor, $inst_1$, the new loop entry. In practice, loops are rotated multiple times in succession in order to move the entry node to a desired position.

This transformation operates on CFG programs, i.e., its input and output are CFG programs. Formally, a loop of an input CFG program G is a triple $(\ell_{pred}, \ell_{entry}, L)$, where L is the set of node labels that form the loop, and ℓ_{pred} is the label of the only node outside of L that has a successor in L, namely the entry point $\ell_{entry} \in L$. For simplicity, we require that the loop predecessor ℓ_{pred} has only ℓ_{entry} as a successor, i.e., it is an instruction node $\ell_{pred} : i \rightarrow \ell_{entry}$. Given a loop $(\ell_{pred}, \ell_{entry}, L)$ to be rotated, the transformation $(\!| G |\!)$ extends the CFG program G with a node labeled ℓ_{copy} which is a copy of the ℓ_{entry} node apart from its label. It also changes the $\ell_{pred} : i \rightarrow \ell_{entry}$ node to $\ell_{pred} : i \rightarrow \ell_{copy}$, so that ℓ_{copy} is executed before the loop.

Theorem 5.8. *Loop Rotation is CT transparent.*

Proof Sketch. We prove the transparency of Loop Rotation with a lock-step simulation and Theorem 4.2. We instantiate the simulation relation $\sim$ so that it holds on $\langle \ell_s, \rho, \mu \rangle \sim \langle \ell_t, \rho, \mu \rangle$, when either $\ell_s = \ell_t$, or $\ell_s = \ell_{entry}$ and $\ell_t = \ell_{copy}$. This means that the register map ρ and memory μ coincide exactly. The observation transformer is the identity. Requirement (i) from Theorem 4.2 holds because program locations also coincide except for when the output program is at label ℓ_{copy}: in that case, the input program is at ℓ_{entry}, which has the same behavior by definition of ℓ_{copy}. Requirements (ii) from Theorem 4.2 and (iii) from Theorem 4.2 hold trivially. □

5.3 Nontransparent Transformations

This section presents the program transformations from Table 2 that do not reflect CT, providing minimal counterexamples to illustrate the kind of removed CT violations removed. In all examples, the input program to the transformation leaks the value of a secret variable `sec` and the transformation removes this CT violation. As discussed in Section 2.2, we used the counterexamples to test real-world decompilers for reflection failures.

If Conversion. If Conversion simplifies the control flow of a program by converting a conditional branch into an equivalent branchless conditional move statement, which are supported by many ISAs. For example, `x86-64` provides the `CMOVcc` instruction to conditionally update a register when a condition holds. Contrary to conditional branches, conditional moves do not leak their condition. Listing 1 presented the counterexample to CT reflection, as explained in Section 2.

Branch Coalescing. This transformation removes conditional branches when their branches are identical. Listing 2 presents an example of Branch Coalescing: the input program leaks the value of `sec` via the conditional branch, but the in output program the branch was coalesced. This means

Listing 2. Branch Coalescing.

```
if (sec) { x = 42; }
else { x = 42; }
```
```
x = 42;
```

Listing 3. Empty Branch Coalescing.

```
if (sec) { } else { }
```

Listing 4. Dead Load Elimination.

```
x = [sec];
x = 42;
```
```
x = 42;
```

Listing 5. Dead Store Elimination.

```
z = [sec];
y = 42;
[sec] = z;
```
```
y = 42;
```

that the output program does not leak sec. A special case of this transformation present in many decompilers is Empty Branch Coalescing (Listing 3), which coalesces only on empty branches.

Memory Access Elimination. This is a family of transformations that remove unnecessary memory accesses. One example from this family is Dead Load Elimination (Listing 4), where a value loaded from memory is never used. Another example is Dead-Store Elimination (Listing 5), where a store is guaranteed to not change the value of its location. In both listings, sec leaks due to a memory access, but the output programs have no such access. Thus, the value is no longer leaked.

6 Improving the Transparency of Practical Decompilers

In this section, we aim to improve a practical decompiler, RetDec, in order to enable better detection of CT violations. We modify RetDec by disabling passes that violate CT transparency. To guide our search for the passes to keep and to discard we leverage our theoretical findings from Section 5. Because RetDec employs a total of 62 passes, we fall back to empirical methods to decide which passes to disable for the passes not covered in Section 5. This means that there are two reasons that our modified version of RetDec may still not be transparent: the empirical tests do not guarantee that passes are transparent, and even the passes shown to be transparent in Section 5 might contain implementation bugs. However, as we will see in our evaluation of the modified version of RetDec, this fact does not diminish the usefulness of improving the transparency of RetDec.

We first give an overview of RetDec, describe the transformations it performs and discuss their coverage by passes from Section 5. We then turn to our empirical methods for other passes. Finally, we combine our modified version of RetDec with CT-LLVM [70], a CT analysis tool for LLVM-IR, to detect CT violations in binaries.

6.1 RetDec Transformations

RetDec is a state-of-the-art machine code decompiler that outputs LLVM IR. It generates well-readable code due to its reliance on LLVM's analysis and optimization passes. We chose to modify RetDec over other decompilers, because it has a well-structured decompilation pass management. All transformation passes are organized in a configuration JSON file. This makes it a good fit to enable or disable decompiler passes at will.

RetDec's decompilation process takes three steps: convert binary code to LLVM IR, optimize LLVM IR, and convert LLVM IR to C code. Overall, RetDec uses 62 distinct passes to decompile and optimize the code: 27 passes are implemented by RetDec itself and the remaining 35 passes are borrowed from LLVM. Most RetDec builtin passes do not perform extensive program transformation. The big exception is the lifting/structural analysis from binary to LLVM IR (*retdec-decoder*), and some minor exceptions being refinement of the decompilation output (e.g., *retdec-simple-types*) or user-configurable passes (e.g., *retdec-select-funcs*). We categorize the remaining LLVM passes into six groups, and discuss, which of them and how they are covered by our theoretical findings

from Section 5. A complete list of all passes, their inclusion in our modified version of RetDec, and which passes are covered by Section 5 is available in Appendix B.

Code Simplification/Elimination. This category contains ten passes that simplify or eliminate code, such as *adce*, *dse*, *early-cse* and *bdce*, which remove dead code. This includes the removal of dead load/store operations and dead branches. We demonstrated that removing dead memory operations makes these passes nontransparent (Listings 4 and 5). One exception is the *strip-dead-prototypes* pass, which removes unused function prototypes. This transformation is similar to Dead Branch Elimination from Section 5, which we proved transparent.

Loop Optimizations. This category contains ten passes that optimize loops. Some passes are used to canonicalize loops or replace them with non-loop forms. For example, *loop-rotate* is used to convert do-while loops to while loops, and *loop-idiom* is used to transform simple loops into a non-loop form (e.g., replace a loop with a `memcpy` call). We have proved in Section 5 that Loop Rotation is transparent. However, some of the other passes are not transparent. For example, RetDec invokes *loop-deletion* to remove loops that have no side-effects to memory and do not contribute to the program output. The removal of such loops can remove CT violations similar to Memory Access Elimination from Section 5.

Expression Substitution. This category contains four passes that perform various expression substitution optimizations. Specifically, RetDec invokes *constprop*, *correlated-propagation* and *scp* to propagate constants, and *constmerge* to merge duplicate constants. These passes are special cases of Expression Substitution from Section 5, which is transparent.

Control Flow Optimization. This category contains four passes that simplify the control flow graph of the program. Two passes, *simplifycfg* and *loop-simplifycfg*, are used to remove unreachable blocks and empty branches, which we have shown to not be transparent (Listing 2). Another pass, *jump-threading*, is used to remove redundant branches, whose conditions are known to be constant at compile time. Removing branches with constant conditions is precisely Dead Branch Elimination from Section 5, which we have proven transparent. Finally, *sink* is used to move instructions to the blocks where they are really used. This pass is not transparent as it could potentially move memory accesses into a dead branch, so the memory access is effectively eliminated (Listings 4 and 5).

Stack Optimization. This category contains one pass, *mem2reg*, that promotes stack variables to register variables. This is similar to our Unspilling pass, which we have proved to be transparent.

LLVM Analysis. This category contains six passes that do not transform the program but provide analysis results to other passes. Since they neither remove nor introduce code, they are transparent.

6.2 Empirical Transparency

We now focus on our empirical methodology to test the transparency of the passes that are not covered by the transformations in Section 5, and present how we modified RetDec.

Methodology. In a nutshell, we empirically test the transparency of RetDec passes on carefully crafted binaries that contain CT violations. Technically, we select which passes are enabled in RetDec, feed it with the binaries, and run CT-LLVM [70] to check for CT violations in the decompiled code. CT-LLVM is a recently published CT analysis tool for LLVM-IR, which we chose due to its precision and usability. In principle, however, any source-level or LLVM-level CT analysis could be used. We call a set of transformation passes *empirically transparent* if the CT analysis tool finds the same CT violations that were contained in the original binaries.

Listing 6. Inverse If Conversion.

```
1
2 mov    rax, 0x3
3 mov    rcx, 0x5
4 cmp    rdi, 0x1
5 cmovne rax, rcx
6 ret
```

```
    cmp rdi, 0x1
    jne T1
    mov rax, 0x3
    ret
T1: mov rax, 0x5
    ret
```

Empirical Test Cases. Our set of empirical test cases checks for both reflection and preservation of the RetDec passes not covered by Section 5. For reflection, we use our test cases from Listings 1 to 5. These cover three common failures to reflection: Branch Coalescing, If Conversion and Dead Load/Store Elimination. For preservation, we construct one test case according to a non-preservation case reported by a recent work [73], which we present in Listing 6. Specifically, this test case checks whether RetDec reverts If Conversion by replacing a conditional move by a conditional branch, which introduces a false positive in CT analysis.

Modifying RetDec. We modify RetDec in two steps. First, we build a minimal version of RetDec containing only version by removing all passes that are not required for RetDec to function. Seven necessary passes remain: *retdec-provider-init*, which is used to initialize the decompiler, *retdec-decoder*, an actual transforming pass, which translates assembly ISA instructions to LLVM-IR instructions, and four passes *retdec-write-ll*, *retdec-write-bc*, *retdec-write-dsm*, and *retdec-llvmir2hll* that are essential for generating outputs. Besides, we keep the *retdec-param-return* pass in the minimal RetDec because it reconstructs function arguments so that CT analyses can be applied. Minimal RetDec is empirically transparent with our test cases.

In order to improve minimal RetDec, we employ an incremental testing strategy. We iteratively enable passes to test their empirical transparency. When we identify a pass that violates CT transparency, we discard it. We also test the passes that we have proven transparent, because a "buggy" implementation could also invalidate our theoretical findings. We do not find any pass that we have shown to be transparent fail any test case, though. Meanwhile, we identify 10 passes that are not transparent. Half of them, *adce*, *dse*, *simplifycfg*, *early-cse* and *bdce*, have already been discussed in Section 6.1. Among the other half, we find that using *globalopt*, *gvn* or *instcombine* removes the empty branch in Listing 3, while using *retdec-inst-opt-rda* or *reassociate* removes the dead load operation in Listing 4. As a result, we discard these ten nontransparent passes.

6.3 Evaluation: Finding CT Violations with CT-RetDec

We use our modified version of RetDec for the Decompile-then-Analyze approach: We build a new binary-level CT analysis tool, called CT-RetDec, which combines the modified RetDec version with the analysis tool CT-LLVM. In this section, we evaluate the performance of CT-RetDec on a benchmark of binaries.

Benchmark Set. We construct the benchmark set with timing side-channel leakages reported by previous works [27, 53, 56, 61]. Specifically, we include the Clangover vulnerability [53], five vulnerabilities in selection algorithms [61], two vulnerabilities in sorting algorithms [27], the check scalar vulnerability in BearSSL [56] and the CMOVNZ vulnerability in HACL* [56]. Our benchmark set takes advantage of the fact that we are employing the Decompile-then-Analyze approach: eight of the listed vulnerabilities are compiler-induced, i.e., the original source code does not exhibit the vulnerability, but only the compiled binary.

We create our benchmark by compiling the source code of the vulnerabilities with Clang in different versions, under different optimization parameters, and different target architectures.

Different from previous works, we compile the source code with newer compiler versions, namely Clang-10, Clang-14, Clang-18 and Clang-21, two optimization levels (no optimization `-O0` and optimization for space `-Os`), and two different platforms (`x86-32` and `x86-64`). This results in 160 distinct binaries to analyze. We then use CT-RetDec to detect which configurations of version, optimization parameters, and target architecture induce the vulnerabilities.

Experiment Results. We evaluated the 160 binaries using CT-RetDec. First, to obtain ground truth, we manually inspected the assembly code of all binaries to establish whether they contain a CT violation. We present the analysis results in Table 3. The prefix `ct_` of the test cases indicates that the source code is CT, while those prefixed `nct_` have a CT violation in the source code. We mark the result of CT-RetDec with ✗ if it reports a CT violation, and with ✓ if it reports no CT violation (the background shades can be ignored for now). We confirm that CT-RetDec correctly find all CT violations while not reporting any false positives. Hence, the results of CT-RetDec match the established ground truth.

The results show that CT-RetDec correctly captures the difference among different compiler versions. For example, it shows that the Clangover vulnerability is not present in binaries compiled with Clang-10 and Clang-14, while it is present in those compiled with Clang-18 and Clang-20. Second, CT-RetDec correctly captures the difference between 32-bit and 64-bit binaries. For example, it shows the constant-time selection algorithms (e.g., `ct_select_v1`) are always constant-time for 64-bit binaries, but not for 32-bit binaries when using `-Os` optimization. This is likely due to architecture-specific optimizations performed by the compiler. Third, CT-RetDec is precise: for all the CT binaries, CT-RetDec preserves CT to the decompiled program. For example, it shows the vulnerabilities in `nct_select` and `nct_sort` disappear in 64-bit binaries obtained with `-Os` optimizations. These vulnerabilities disappear because compiler replaces the conditional branches with conditional moves, which are constant-time. To summarize, we show that CT-RetDec is effective in analyzing binaries and finding compiler-induced CT violations.

Comparison with Unmodified RetDec+CT-LLVM. We now show that CT-RetDec outperforms the combination of unmodified RetDec and CT-LLVM. Specifically, we find that the nontransparent RetDec does miss most CT violations in the benchmark set. We make the comparison by running the unmodified RetDec + CT-LLVM on the benchmark set. We highlighted all CT violations missed by nontransparent RetDec in Table 3 with a shaded green cell ✗. As we can see, it can only find leakages in the sorting algorithms, `ct_check_scalar` and `ct_hacl_cmovzn4` under some configurations. Our comparison results highlight the importance of transparent decompilation for finding CT violations.

Performance Overhead of CT-RetDec. By selectively disabling optimization passes for CT-RetDec, we can expect increases in both the size of the decompiled code and the cost of subsequent analysis. On average, using CT-RetDec incurs a 73% increase in analysis time and a 316% increase in decompiled code size. Figure 7 summarizes these overheads across different binary types and compiler optimization levels. Binaries compiled with `-O0` have the largest overhead, as they contain more unoptimized code and therefore offer greater opportunities for decompilation optimizations that are disabled in CT-RetDec. In contrast, binaries compiled with `-Os` are already compact and leave less room for further optimization, resulting in lower performance overhead. Given the benefit of using a more transparent RetDec, this performance overhead is generally acceptable.

7 Nontransparent Transformations in CT Analysis Tools

So far, we have focused on the Decompile-then-Analyze approach, ensuring that all program transformations applied by the decompiler previous to calling the CT analysis tool are transparent.

Table 3. CT-RetDec analysis results. CT-RetDEc correctly finds all CT violations (marked with ✗) and does not report false positives for CT binaries (marked with ✓).

	Clang-10.0.0				Clang-14.0.6				Clang-18.1.8				Clang-21.1.1			
	64-bit		32-bit		64-bit		32-bit		64-bit		32-bit		64-bit		32-bit	
Test case	O0	Os	O0	Os	O0	Os	O0	Os	O0	Os	O0	Os	O0	Os	O0	Os
ct_clangover	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗
ct_select_v1	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
ct_select_v2	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
ct_select_v3	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
ct_select_v4	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
ct_sort	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✗
ct_check_scalar	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗
ct_hacl_cmovzn4	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
nct_select	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
nct_sort	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗

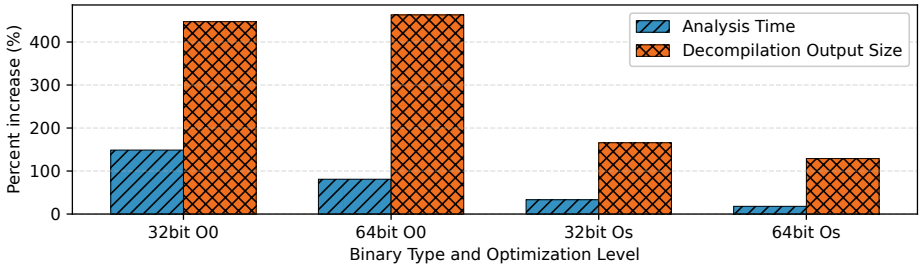


Fig. 7. Performance Overhead of CT-RetDec compared to the combination of unmodified RetDec+CT-LLVM.

However, there is a caveat: Many CT analysis tools start their own analysis by also transforming programs into an alternative representation previous to the actual analysis. Table 4 provides a partial list of CT tools, indicating for each of them their input language, the language on which the analysis is carried out, and which tool is used for the conversion. CacheS [66] implicitly uses the Decompile-then-Analyze approach as it integrates the BINNAVI decompiler to decompile binaries to REIL IR before applying the CT analysis. Similarly, BinSec [27] and BinSec/Haunted [28] internally lift binary programs to DBA IR, which is an IR close to but richer than assembly language, before performing CT analysis. To ensure the soundness of the listed CT analysis tools, they must guarantee that their employed converters are transparent. In this section, we examine the converters used by CT-Verif and BinSec.

7.1 CT-Verif

CT-Verif [6] is a state-of-the-art CT analysis tool for LLVM-IR programs. CT-Verif is based on a product program construction, which is sound (non-CT programs are never deemed CT) and relatively complete (under certain conditions, CT programs never deemed non-CT).

Conversion. CT-Verif [6] internally uses SMACK [54] to transform an LLVM IR program P into a Boogie program $(\!|P|\!)$ that emulates two lock-step executions of P. The emulation additionally contains assertions that check for differing leakage in the two executions to detect CT violations.

Table 4. Program transformations in CT analysis tools.

CT Analysis Tool	Input Language	Output Language	Converter
FlowTracker [55]	C	LLVM IR	LLVM
CANAL [62]	C	LLVM IR	LLVM
Disselkoen et al. [29]	C	LLVM IR	LLVM
Barthe et al. [12]	C	MachIR	CompCert
Blazy et al. [19]	C	C#minor	CompCert
ctverif [6]	LLVM IR	Boogie IR	SMACK
SideTrail [10]	LLVM IR	Boogie IR	SMACK
Cai et al. [23]	LLVM IR	Boogie IR	SMACK
CacheS [66]	Binary	REIL	BINNAVI
BinSec/Rel [27]	Binary	DBA	BinSec
BinSec/Haunted [28]	Binary	DBA	BinSec

Listing 7 Non-CT program accepted by CT-Verif.

```
1 int public_array[2] = {100, 101}; C Code
2 int func(int secret_bit) {
3     int dead_load = array[secret_bit];
4     return 1;
5 }
```

```
   LLVM IR
define i32 @func(i32) {

    ret i32 1;
}
```

The SMACK-based transformation from LLVM IR to Boogie performs standard optimizations at IR level, including Dead Load/Store Elimination. Unfortunately, Dead Load/Store Elimination is not transparent (Listings 4 and 5). This makes the results of making it possible to craft an example of a non-CT `C` program that is accepted by CT-Verif, a soundness bug.

Experiment. To confirm our hypothesis, we construct such a program and apply CT-Verif to analyze it. The program is shown on the left side of Listing 7. It violates CT because the secret input `secret_bit` is used as a memory address (Line 3). However, the CT violation is removed by Dead Code Elimination in SMACK and no longer exists in the code shown on the right side of Listing 7 on which the safety analysis is performed. Therefore, the transformation does not reflect CT, and CT-Verif erroneously reports that the program is constant-time.

7.2 BinSec

BinSec [27] is a state-of-the-art binary-level CT analysis tool based on bounded verification with relational symbolic execution, which is also sound and relatively complete.

Conversion. BinSec lifts a binary program to an IR called DBA (for Dynamic Bitvector Automata) before it performs relational symbolic execution on the DBA program. When lifting a binary program to DBA IR, BinSec converts conditional branches into two `goto` statements—one points to the target address and the other to the fall-through address. However, the lifting also performs Empty Branch Coalescing (Listing 3), which we have shown to be nontransparent, a soundness bug in BinSec.

Experiment. We confirm our hypothesis that BinSec can accept non-CT programs by constructing the program shown on the left side of Listing 8. The program compares a secret input argument `rdi` to one. If the two are equal, it branches to address `T1`. Otherwise, it executes the fall-through code, which is also at `T1`. This binary violates the CT policy because it branches on a secret.

Listing 8 Non-CT program accepted by BinSec.

```
1   cmp rdi, 1          ASM          cmp rdi, 1        DBA IR
2   je T1                            goto T1
3 T1:                              T1:
4   mov rax, 1                       mov rax, 1
```

BinSec converts the program to the DBA IR shown on the right side of Listing 8. Critically, the DBA program contains an unconditional `goto` statement but no longer branches on a secret, which means that the CT violation is gone. BinSec erroneously reports that the input program is constant-time, a soundness bug.

Discussion. Our finding is unlikely to invalidate the results of BinSec on real-world libraries, as conditionals with two empty branches are unlikely to exist in the real-world. However, the gap between theory and practice caused by nontransparent transformations could be exploited by a malicious agent that could intentionally introduce such examples into a popular library.

A separate question is whether the synthetic pattern from Listing 8 could be exploited in practice. The answer is that it can, using the BunnyHop technique [71]. Specifically, an adversary can monitor the branch predictor state, which is updated after each branch execution, to infer whether a secret-dependent branch was taken. We confirm the feasibility of such an attack by modifying the artifact from [71] to use our synthetic example as the victim. The details of the attack and the experimental results are provided in Appendix C.

8 Transparency for Speculative Constant-Time

Constant-time programs remain vulnerable to Spectre attacks [43], where an attacker manipulates speculative execution mechanisms (e.g., triggering branch mispredictions) to recover secrets from speculative leakage. A common way to protect programs against Spectre attacks is to follow the speculative constant-time (SCT) policy [14, 26], which extends the constant-time policy to the speculative setting—the literature presents a few variants of this policy, e.g., [36]. There exists various tools [25, 26, 28, 36, 59, 64] to check SCT—or its variants. This raises the question of whether we can lift our results to verify a transformation is SCT transparent.

This section discusses how to extend our results to SCT transparency. We first show how to adapt our definitions of preservation, reflection, and transparency to SCT. We then lift simulation diagrams and their soundness results to SCT (leaving full details to Appendix D). Finally, we summarize our results analyzing several standard transformations for SCT transparency.

8.1 Speculative Constant-Time Transparency

Speculative semantics such as [14, 59] instrument the semantics of languages with *directives* $\mathcal{D}$, to capture the effect of branch prediction, and the fact that the attacker can poison said predictor. We follow this direction and adapt our language model from Section 3 to a speculative semantics with branch prediction. Each step in the semantics is guided by a directive $d \in \mathcal{D}$; $s \xrightarrow[d]{o} s'$ means that s steps under directive d to s' and produces the observation o. We consider two directives: step and force, which model correct and incorrect predictions, respectively. We use the step directive also for steps that require no prediction, such as assignments. We present the semantics in Figure 8.

Definitions 3.1 to 3.3 readily generalize to the speculative setting. Specifically, the speculative behavior of a program collects all executions possible under any sequence of directives (this formulation is inspired by [63]).

Definition 8.1. The speculative behavior of P on an input i is $SBeh(P, i) \triangleq \{(\boldsymbol{d}, \boldsymbol{o}) \mid P(i) \xrightarrow[\boldsymbol{d}]{\boldsymbol{o}}{}^{*} s\}$.

$$\textsc{Step}\ \frac{s \xrightarrow{o} s'}{s \xrightarrow[\text{step}]{o} s'} \qquad \textsc{If}\ \frac{b = [\![e]\!]_\rho}{\langle \texttt{if } e \texttt{ then } c_{\texttt{tt}} \texttt{ else } c_{\texttt{ff}}, \rho, \mu\rangle \xrightarrow[\text{force}]{\text{br } b} \langle c_{\neg b}, \rho, \mu\rangle} \qquad \textsc{While}\ \frac{[\![e]\!]_\rho = b \quad c_{\texttt{ff}} = \texttt{skip} \quad c_{\texttt{tt}} = c; \texttt{while } e \texttt{ do } c}{\langle \texttt{while } e \texttt{ do } c, \rho, \mu\rangle \xrightarrow[\text{force}]{\text{br } b} \langle c_{\neg b}, \rho, \mu\rangle}$$

Fig. 8. Speculative Semantics.

Table 5. Summary of decompilation passes analyzed for SCT-transparency. A ✓ means that we have proven the pass SCT-transparent using an SCT-simulation, a ↯ that we found a counterexample to SCT reflection, and • means the pass is SCT preserving, but we omit the proof.

Pass	SCT-Reflection	SCT-Preservation
Constant Folding	✓	✓
Structural Analysis	✓	✓
Dead Branch Elimination	↯	•
Dead Assignment Elimination	↯	•
Unspilling	↯	•

A program is speculative constant-time w.r.t. an indistinguishability relation ϕ, denoted ϕ-SCT, when for all sequences of directives, related inputs yield equal observations.

Definition 8.2 (ϕ-SCT). P is ϕ-SCT if for all inputs i_1 and i_2, $i_1 \ \phi \ i_2 \implies \mathit{SBeh}(P, i_1) = \mathit{SBeh}(P, i_2)$.

Reflection, preservation, and transparency are defined analogously to Definition 3.3.

Definition 8.3 (Reflection, Preservation, and Transparency for SCT). We say that a program transformation $(\!|\cdot|\!) : \mathcal{L}_s \to \mathcal{L}_t$ between an input language $\mathcal{L}_s$ and an output language $\mathcal{L}_t$

- Reflects SCT if for each P and ϕ, $(\!|P|\!)$ is ϕ-SCT implies P is ϕ-SCT;
- Preserves SCT if for each P and ϕ, P is ϕ-SCT implies $(\!|P|\!)$ is ϕ-SCT; and
- Is SCT transparent if it both reflects and preserves SCT.

8.2 Proof Techniques

We can extend the CT simulations from Definition 4.3 to the speculative setting, as well as Theorem 4.5, by accounting for the directives in the speculative semantics. The key idea is to obtain a bisimulation by linking the directives of the input program and the output program. For that, we employ *directive transformers* (introduced in [9]), which function analogously to observation transformers from Section 4. They are functions $T_{\mathcal{D}} : \mathcal{PC} \to (\mathcal{D}_t \to \mathcal{D}_s)$ that translate the directives for the output program into directives for the input program and vice versa ($T^{\text{pc}}_{\mathcal{D}}$ is bijective for all $\text{pc} \in \mathcal{PC}$). Given two states in simulation $s \sim t$ and a step $t \xrightarrow[d]{o} t'$, the directive transformer computes a directive $T_{\mathcal{D}}(\text{pc}(s), d)$ to determine the simulating step from s. Appendix D.1 includes the detailed definitions of these simulations.

8.3 Summary of Results

We studied the SCT transparency of five standard transformations and summarize the results in Table 5. Appendix D.2 applies our simulations to Structural Analysis (as described in Section 5) and Constant Folding (a special case of Expression Substitution from Section 5) to prove them SCT transparent. On the other hand, Appendix D.3 shows that the other transformations are not SCT

transparent, as we summarize next. Dead Branch Elimination is not SCT transparent because the body of a dead branch could be executed speculatively and still leak a secret. Similarly, in Dead Assignment Elimination, an assignment could be considered as dead if it affects only instructions within a dead branch. However, during speculative execution, that branch could be executed speculatively, leading to leakage of a secret propagated by the dead assignment. Unspilling is not SCT transparent because data stored on the stack could be speculatively accessed via out-of-bounds access and leaked later. Unspilling would move such data from the stack to registers, which makes it safe from speculative out-of-bounds accesses.

9 Related Work

Our work draws on the secure compilation literature to address the question of whether decompiler transformations undermine the soundness or accuracy of side-channel analysis. Accordingly, we divide this section into three parts: first, we overview side-channel analysis and decompiler transformations; then, we review related work in the field of secure compilation; finally, we briefly review related work on decompilation and discuss its relevance.

Side-Channel Analysis. Barbosa et al. [11], Geimer et al. [35], and Jancar et al. [42] survey a large number of tools to detect side-channel vulnerabilities. Several of these tools build on robust theoretical foundations, such as product programs [6], type systems [5], SMT solvers [20], abstract interpretation [19], and symbolic execution [27, 29]. All of these, and the majority of the ones in the survey, operate on source code or an IR, which means that applying them to assembly programs requires decompiler transformations. Section 7 overviews different tools and discusses CT-Verif [6] and BinSec [27] in depth.

Secure Compilation. It is well known that secure source programs can be transformed into insecure binary programs, leading to a dangerous *compiler security gap* [31, 61, 67]. The field of secure compilation aims to close this gap by integrating security considerations into compiler development [3, 4]. Consequently, we draw on this body of work to recast existing techniques for preservation of side-channel security into a rigorous approach for CT transparency.

Preservation of CT was first considered in [15]. The same work introduces CT simulations as the primary tool for proving preservation. Later work [16] uses the equivalent but simpler technique of leakage transformers. Both the Jasmin compiler and (a mild modification of) the CompCert compiler are shown to preserve constant-time in [13] and [16], respectively. Our notion of simulation uses a variant of CT simulation that adds the $\mathcal{PC}$-injectivity condition on leakage transformers.

Decompilation in Program Analysis. A number of prior works study the correctness of decompilers [21, 22, 57, 65, 74]. However, they focus on the input/output behavior of programs and yield no guarantees about the security impact of decompilers. Previous work explores and, to a large extent, confirms the potential of source analysis of decompiled programs [47, 48, 72]. Nevertheless, they neither formalize nor provide proof techniques for secure decompilation.

Liu et al. [47] explore the impact of lifters on pointer analysis, and discriminability analysis in the context of LLVM lifters. Their approach is empirical, comparing two LLVM IR representations of a source program. The first is obtained by compiling from source to LLVM IR, and the second by compiling the program to a binary and subsequently lifting the binary to LLVM IR.

Mantovani et al. [48] perform a complementary exploration. They consider buffer overflows, integer overflows, null pointer dereference, double free or use after free, and division by zero. In addition, they provide several recommendations for decompiler writers. One of their recommendations resonates strongly with the findings of our work: departing from the human-centric view of decompilers in favor of improving the soundness and completeness of decompilers.

10 Future Work

In this section, we discuss several possible directions for future work.

Mitigation Strategies. In this paper, we mitigate the transparency issue in a real-world decompiler by finding and disabling nontransparent transformations. Besides this approach, it is possible to add marks to force the decompiler to not optimize certain parts of the code. This would be similar to how cryptographic developers mark secrets with the *volatile* keyword to hint to the compiler not to optimize operations that process them [40, 61]. The benefit of this approach is that decompiler optimizations can still be run on unmarked code, which reduces the size and improves the readability of decompiled code. The downside is that the users of the decompiler may need to manually identify cases that require the marks, which could be time-consuming and error-prone. Furthermore, ensuring that decompilers support such marks robustly and reliably is far from trivial.

Another possible mitigation strategy is to extend the IR/high-level-language with a *leak* primitive, a no-op instruction whose only purpose is to leak a specific value. It allows the decompiler to optimize the code as usual while inserting the *leak* instruction to indicate that a value could be leaked at that point. Again, implementing such a mitigation strategy in decompilers is nontrivial. We leave the investigation of these mitigation strategies to future work.

Generalization. Since we primarily focus on the RetDec decompiler in this paper, one natural future research direction is to generalize our findings to other popular decompilers. Specifically, it would be interesting to locate their nontransparent transformations and develop patches or extensions to control or disable them. The investigation could also be extended to the semantic gap between binaries and decompiled code, and differences in undefined behavior, in order to identify their impact on transparency. Besides finding and mitigating nontransparent transformations in decompilers, it would also be valuable to build a more comprehensive benchmark set to empirically evaluate the transparency of decompilers and the effectiveness of the mitigations.

Transparency for More Transformations. Apart from decompilation, an interesting future direction would be to study the transparency of program transformations for hardware-software contracts [37], a general framework that encompasses different leakage and execution models. More broadly, it would be interesting to study transparency of program transformations for other classes of security-related properties, e.g., memory properties. Finally, it would be interesting to develop techniques to prevent decompilers to eliminate potentially harmful code—similar to techniques for preventing compilers to introduce harmful code.

11 Conclusion

In this paper, we initiate the study of CT transparency. We prove that state-of-the-art decompilers remove CT violations and correct this issue with the tool CT-RetDec, which transparently decompiles our benchmark set of 160 binaries. We provide rigorous proof methods to assert the transparency of transformations, and demonstrate them on common transformations. Our work also emphasizes the need for developers of constant-time verification tools to ensure transparency *within* their tools—specifically in the transformations that convert the input program into the representation used for analysis. Our recommendations for CT tool developers are: first, extensively test for transparency of the initial transformations; second, carefully document the initial transformations; third, if possible, expose these transformations for scrutiny, ideally by clearly separating the transformation phases and the analysis phases in the source code of the tool.

Acknowledgments

We thank the reviewers for their time and insightful feedback. This research was supported by the *Deutsche Forschungsgemeinschaft* (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972.

Data-Availability Statement

This work provides an artifact containing:

- A Rocq formalization of a general version of Theorem 4.5;
- The vulnerable Clangover binary and decompiled code;
- Test cases we use to test state-of-the-art decompilers and the presented results;
- Original, minimal, and final RetDec configurations;
- Code to test if RetDec is empirically transparent and the presented results;
- Experiments to find CT violations with CT-RetDec and the presented results; and
- Two vulnerable test cases for CT-Verif and BinSec.

The artifact is available at [8] and was submitted for Artifact Evaluation.

References

[1] Vector 35. 2016. Binary Ninja. https://binary.ninja/
[2] Vector 35. 2025. *Dogbolt.* https://dogbolt.org/
[3] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 14:1–14:48. doi:10.1145/3460860
[4] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 256–271. doi:10.1109/CSF.2019.00025
[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. doi:10.1145/3133956.3134078
[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*. 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida
[7] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 583–600. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse
[8] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Youcef Bouzid, Sören van der Wall, and Zhiyuan Zhang. 2026. *Artifact for Paper Decompiling for Constant-Time Analysis.* doi:10.5281/zenodo.18749373
[9] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2025. Preservation of Speculative Constant-Time by Compilation. *Proc. ACM Program. Lang.* 9, POPL, Article 44 (Jan. 2025), 33 pages. doi:10.1145/3704880
[10] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2018. Sidetrail: Verifying time-balancing of cryptosystems. In *VSTTE*. 215–228.
[11] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *SP*. 777–795. doi:10.1109/SP40001.2021.00008
[12] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *CCS*. 1267–1279.
[13] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. doi:10.1145/3371075
[14] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1884–1901. doi:10.1109/SP40001.2021.00046

[15] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 328–343. doi:10.1109/CSF.2018.00031

[16] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 462–476. doi:10.1145/3460120.3484761

[17] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity24/presentation/basque

[18] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[19] Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying Constant-Time Implementations by Abstract Interpretation. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10492)*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer, 260–277. doi:10.1007/978-3-319-66402-6_16

[20] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[21] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *USENIX Security*. 353–368. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[22] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. 2022. Decomperson: How Humans Decompile and What We Can Learn From It. In *USENIX Security*. 2765–2782. https://www.usenix.org/conference/usenixsecurity22/presentation/burk

[23] Luwei Cai, Fu Song, and Taolue Chen. 2024. Towards Efficient Verification of Constant-Time Cryptographic Implementations. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1019–1042.

[24] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. 2024. Evaluating the Effectiveness of Decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 491–502. doi:10.1145/3650212.3652144

[25] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *SP*. 666–680. doi:10.1109/SP46214.2022.9833707

[26] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 913–926. doi:10.1145/3385412.3385970

[27] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *SP*. 1021–1038. doi:10.1109/SP40000.2020.00074

[28] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *NDSS*. https://www.ndss-symposium.org/ndss-paper/hunting-the-haunter-efficient-relational-symbolic-execution-for-spectre-with-haunted-relse/

[29] Craig Disselkoen, Sunjay Cauligi, Dean Tullsen, and Deian Stefan. 2020. Finding and eliminating timing side-channels in crypto code with pitchfork. In *TECHCON*.

[30] Luke Dramko, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A Taxonomy of C Decompiler Fidelity Issues. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity24/presentation/dramko

[31] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 73–87. doi:10.1109/SPW.2015.33

[32] Steffen Enders, Eva-Maria C. Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. 2022. dewolf: Improving Decompilation by leveraging User Surveys. *CoRR* abs/2205.06719 (2022). arXiv:2205.06719 doi:10.48550/ARXIV.2205.06719

[33] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. 2011. Enhanced structural analysis for C code reconstruction from IR code. In *14th International Workshop on Software and Compilers for Embedded Systems, SCOPES '11, St. Goar, Germany, June 27-28, 2011*, Henk Corporaal and Sander Stuijk (Eds.). ACM, 21–27.

doi:10.1145/1988932.1988936

[34] Antoine Geimer and Clementine Maurice. 2025. Fun with flags: How Compilers Break and Fix Constant-Time Code. *arXiv preprint arXiv:2507.06112* (2025).

[35] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *CCS*. ACM, 1690–1704. doi:10.1145/3576915.3623112

[36] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1–19. doi:10.1109/SP40000.2020.00011

[37] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1868–1883. doi:10.1109/SP40001.2021.00036

[38] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese (Eds.). ACM, 637–651. doi:10.1145/3320269.3384766

[39] Hex-Rays. 2023. Ida Pro. https://www.hex-rays.com/products/ida

[40] Intel. 2022. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations.* https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html Accessed: 2026-01-22.

[41] Intel. 2023. *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 Accessed: 2025-10-06.

[42] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 632–649. doi:10.1109/SP46214.2022.9833713

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE SP*. 1–19. doi:10.1109/SP.2019.00002

[44] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO'96 (LNCS, Vol. 1109)*. SV, 104–113. http://www.cryptography.com/public/pdf/TimingAttacks.pdf

[45] Jakub Křoustek, Peter Matula, and Petr Zemek. 2017. Retdec: An open-source machine-code decompiler. https://github.com/avast/retdec

[46] Zhibo Liu and Shuai Wang. 2020. How far we have come: testing decompilation correctness of C decompilers. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 475–487. doi:10.1145/3395363.3397370

[47] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1100–1119. doi:10.1109/SP46214.2022.9833799

[48] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study. In *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako (Eds.). ACM, 602–615. doi:10.1145/3488932.3497764

[49] James Mattei, Madeline McLaughlin, Samantha Katcher, and Daniel Votipka. 2022. A Qualitative Evaluation of Reverse Engineering Tool Usability. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. ACM, 619–631. doi:10.1145/3564625.3567993

[50] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3935)*, Dongho Won and Seungjoo Kim (Eds.). Springer, 156–168. doi:10.1007/11734727_14

[51] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. 2025. Automated Side-Channel Analysis of Cryptographic Protocol Implementations. *CoRR* abs/2511.11385 (2025). doi:10.48550/ARXIV.2511.11385

[52] National Security Agency (NSA). 2018. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/

[53] Antoon Purnal. 2024. *Clangover CVE.* https://nvd.nist.gov/vuln/detail/CVE-2024-37880 Accessed: 2025-10-03.

[54] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 106–113. doi:10.1007/978-3-319-08867-9_7

[55] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *CC*. 110–120. doi:10.1145/2892208.2892230

[56] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. 2024. Breaking Bad: How Compilers Break Constant-Time Implementations. arXiv:2410.13489 [cs.CR] https://arxiv.org/abs/2410.13489

[57] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*.

[58] Edward J. Schwartz and JongHyup Lee. 2013. Decompilation of Binary Programs Using Control-Flow Structuring and Semantic-Preserving Transformations. In *USENIX Security*. 369–384. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[59] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1094–1111. doi:10.1109/SP46215.2023.10179418

[60] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *SP*. 138–157. doi:10.1109/SP.2016.17

[61] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 1–15. doi:10.1109/EuroSP.2018.00009

[62] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 904–907. doi:10.1145/3238147.3240485

[63] Sören van der Wall and Roland Meyer. 2025. SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations. *Proc. ACM Program. Lang.* 9, POPL, Article 51 (Jan. 2025), 30 pages. doi:10.1145/3704887

[64] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434330

[65] Freek Verbeek, Joshua A. Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally verified lifting of C-compiled x86-64 binaries. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 934–949. doi:10.1145/3519939.3523702

[66] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *USENIX Security*. 657–674. https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai

[67] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3655–3672. https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao

[68] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 158–177.

[69] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *NDSS*. https://www.ndss-symposium.org/ndss2015/no-more-gotos-decompilation-using-pattern-independent-control-flow-structuring-and-semantics

[70] Zhiyuan Zhang and Gilles Barthe. 2025. CT-LLVM: Automatic Large-Scale Constant-Time Analysis. Cryptology ePrint Archive, Paper 2025/338. https://eprint.iacr.org/2025/338

[71] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. BunnyHop: Exploiting the Instruction Prefetcher. In *USENIX Security*. 7321–7337. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-bunnyhop

[72] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *Proceedings of the 29th ACM International Conference on Architectural Support*

for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 912–928. doi:10.1145/3620665.3640382

[73] Quan Zhou, Sixuan Dang, and Danfeng Zhang. 2024. CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming. In *ECOOP (LIPIcs, Vol. 313)*. 46:1–46:26. doi:10.4230/LIPICS.ECOOP.2024.46

[74] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. 2024. D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/zou

Received 2025-10-10; accepted 2026-02-17