



Smooth, Integrated Proofs of Cryptographic Constant Time for Nondeterministic Programs and Compilers

OWEN CONOLY, Massachusetts Institute of Technology, USA

ANDRES ERBSEN, Google, USA

ADAM CHLIPALA, Massachusetts Institute of Technology, USA

Formal verification of software and compilers has been used to rule out large classes of security-critical issues, but risk of unintentional information leakage has received much less consideration. It is a key requirement for formal specifications to leave some details of a system's behavior unspecified so that future implementation changes can be accommodated, and yet it is nonetheless expected that these choices would not be made based on confidential information the system handles. This paper formalizes that notion using omnisemantics and plain single-copy assertions, giving for the first time a specification of what it means for a nondeterministic program to be constant-time or more generally to avoid leaking (a part of) its inputs. We use this theory to prove data-leak-free execution of core cryptographic routines compiled from Bedrock2 C to RISC-V machine code, showing that the smooth specification and proof experience omnisemantics provides for nondeterminism extends to constant-time properties in the same setting. We also study variants of the key program-compiler contract, highlighting pitfalls of tempting simplifications and subtle consequences of how inputs to nondeterministic choices are constrained. Our results are backed by modular program-logic and compiler-correctness theorems, and they integrate into a neat end-to-end theorem in the Coq proof assistant.

CCS Concepts: • **Software and its engineering** → **Formal software verification; Compilers**; • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: cryptographic constant time, compiler verification, omnisemantics

ACM Reference Format:

Owen Conoly, Andres Erbsen, and Adam Chlipala. 2025. Smooth, Integrated Proofs of Cryptographic Constant Time for Nondeterministic Programs and Compilers. *Proc. ACM Program. Lang.* 9, PLDI, Article 215 (June 2025), 24 pages. <https://doi.org/10.1145/3729318>

1 Introduction

Cryptographic software relies on compiler correctness beyond what is captured by a conventional trace-inclusion or specification-preservation result. The crux is that while a number of implementation choices are intentionally left to the compiler, it would not be fair for the compiler to generate code that makes these choices based on examination of confidential data handled by the program. This consideration applies both to data flows that are commonly modeled in compiler-correctness theorems—for example, when iteration order of collections depends on addresses of stack-allocated objects—and also to “side” channels that are often omitted, for example utilization of CPU resources. In either case, it would be unreasonably burdensome for the source-language semantics to specify these details fully, and actually preserving them in the compiled code would severely limit implementation flexibility, completely prohibiting optimization if this idea is taken literally.

Authors' Contact Information: [Owen Conoly](#), Massachusetts Institute of Technology, Cambridge, USA, owenc@mit.edu; [Andres Erbsen](#), Google, Cambridge, USA, andreser@mit.edu; [Adam Chlipala](#), Massachusetts Institute of Technology, Cambridge, USA, adamc@csail.mit.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/6-ART215

<https://doi.org/10.1145/3729318>

Past study [4] of confidentiality-preserving compilation in the context of CompCert [20] leaves these issues out-of-scope: “we assume that the languages are deterministic,” “it does not seem straightforward what it would mean for a non-deterministic C program to be constant-time.” Impressive progress (porting of 17 out of 20 compiler passes) was possible in spite of leaving these fundamental questions unanswered because CompCert’s proof strategy already relies on an intricate deterministic memory model.

We present our development of a new generalization of cryptographic constant time to support nondeterminism. We implement and evaluate this generalization through a full verified software stack in Coq, including program-logic proofs of C-like programs and verified compilation to machine code, with derived timing-independence theorems at the machine-code level. What is most distinctive about our approach is recognition that rules about which program inputs may influence which nondeterministic choices are inherently program-specific, and so we need an expressive specification formalism that pairs well with a compiler proved to preserve specifications. We study several candidate formalisms, summarize their pros and cons, and relate them formally. Every fact presented as a theorem, corollary, or lemma in this paper has been formalized in Coq.

Our full paper, including appendices, is available on the arXiv [12].

1.1 Constant-Time Programming

Let us begin by reviewing the established programming discipline known as *cryptographic constant time*. First, despite the name, this discipline enforces a data-flow property, not a statement about measured or asymptotic execution time. The vast majority of important cryptographic implementations follow this discipline to avoid leaking secrets (e.g. secret keys or plaintexts) through side channels such as timing of program execution, cache occupancy, or CPU resource usage. The same discipline can also be captured using straightforward operational rules specifying information leaked by execution steps. The standard approach that we follow is to apply an instrumented semantics that, as a program runs, accumulates a *leakage trace* of the following kinds of events: read memory at address a , wrote memory at address a , or resolved a conditional jump (given as a test expression’s Boolean value). Note that we log memory *addresses* but not the values read/written, and that the branch decisions being leaked allows control flow to be reconstructed from the program and its leakage trace. If a deterministic program’s input is partitioned into secret and public components, the program is said to be constant-time if, on any two executions *that agree on values of all public inputs*, those two executions *generate identical leakage traces*. It is apparent that constant time generalizes noninterference, where we consider leakage traces as implicit public outputs.

This condition composes well with the vast majority of important compiler and hardware optimizations, with optimizations that are exceptions to this pattern becoming known as microarchitectural side-channel vulnerabilities. For instance, all common systems of memory caching retain secret-independent timing with constant-time programs if configured appropriately, since their “control flow” (e.g., interactions between caches in a cache-coherence protocol) only changes based on memory addresses accessed, not values read/written. However, it is apparent that the definition we just presented no longer captures the desired concept in the presence of nondeterminism. Calling a function twice on *the same* arguments may yield different leakage traces, so we have little hope that calls with different arguments will always generate identical traces. Yet programs expected to be side channel-secure often *do* depend on nondeterminism, whether as a consequence of engineering abstraction (e.g. memory-management details are left unspecified at the source level) or inherently through I/O (e.g. to implement cryptographic protocols, generate randomness, or access long-term keys). Key-and-input-dependent variation in response (output) time has been exploited to perform padding-oracle attacks even against systems that do not deliberately disclose

information about unauthenticated decryption results [1], and constant-time techniques to avoid this issue rely on stack-allocated temporary buffers to store sensitive intermediate values.¹

1.2 Our Semantics Approach in One Page

All source-language and machine-code programs in this paper will be analyzed in terms of which specifications they admit in the sense of total-correctness Hoare logic or omnisemantics [10] (a variation of weakest-precondition calculus that we explain more as we reference aspects of it later). In symbols, $p \Downarrow Q$ means that program p (which we in this section treat as including all relevant initial state) terminates in a state satisfying the predicate Q , and it does so *regardless of the outcomes of nondeterministic choices* during its execution. Note that the set Q is just a constraint on the possible behaviors, and there is no requirement that every outcome in Q is even possible, let alone as likely to occur as others. Thus, we allow compilers to shrink Q , perhaps by compiling a nondeterministic source program to a deterministic machine-code program.

Side-channel leakage is encoded as yet another component of the state of a program, and Q is asserted on it upon termination. For specific programs p , we pick concrete specifications Q that relate the runtime input-output log t of each execution to its leakage trace k . For example, saying that a username u may be leaked but a passphrase s may not: $\exists f. p \Downarrow \{(t, k) \mid \exists u, s. t = [\text{IN } u; \text{IN } s] \wedge k = f(u)\}$. Here the I/O log t is constrained to record inputting just u and s , in that order. Critically, quantifier ordering requires the program to satisfy this specification using one and the same leakage-prediction function f for all passphrases s . If f were existentially quantified after s , it could be the constant function that returns s , allowing the passphrase to be leaked.

The example just presented shows how to encode a leakage specification as a “single-copy” property: a postcondition that all possible executions must satisfy individually. The ability to capture properties that might also be stated in a “two-copy,” hyperproperty style (e.g., saying that for every pair of possible executions, their leakage traces are related) arises from the ability to share quantifiers between the postconditions of the possible executions: we can say that there must exist a reference leakage trace that every possible leakage trace is related to. For example, the example above says that every leakage trace is related to the same function f . Encoding leakage specifications as single-copy rather than two-copy properties greatly simplifies our formalization.

We formalize compiler correctness in the style of specification preservation. Specifically, an optimization pass C that does not change the language or the possible leakage must satisfy $\forall p, Q. p \Downarrow Q \implies C(p) \Downarrow Q$. Compiler passes that change the language or the program state (including leakage) would use different variants of \Downarrow and Q in the conclusion. A common pattern is to define the lower-level postcondition Q' in terms of a state-representation relation R and the high-level postcondition Q . For example, Q' may assert that there exists a machine word that is stored in a calling convention-specified register and also is a permissible return value of the function according to the high-level postcondition Q . Similarly, the leakage specification of a simple compiler pass can state that every leakage trace of the compiled program equals the output of some (global) function g on some leakage trace of the source program: $\forall p, Q. \exists g. p \Downarrow Q \implies C(p) \Downarrow \{(t, g(k)) \mid Q(t, k)\}$.

The situation is more complicated when program control flow can depend on nondeterministic choices that may be resolved by the compiler, perhaps in a way that depends on some part of the program’s state but not another. We will now describe scenarios where these considerations come up and develop ways to specify how we would want a compiler to handle them.

The end point of our journey will be a combined theory including:

- The **first generalization of cryptographic constant time to support nondeterminism**,
- that can specify **which nondeterministic choices may be resolved based on secrets**,

¹For example, see BoringSSL function `EVP_t1s_cbc_copy_mac`.

- which we then use for **verification of multipass compilers that maintain constant time as they reduce some but not all nondeterminism** as we progress through compilation,
- and to give **compiler proof-compatible Hoare-logic certificates to specific C programs representative of constant time-programming challenges**,
- all evaluated through a **Coq development**.

1.3 Overview of the Paper

Section 3.1 describes the basic idea of our approach: to formalize “program p is constant-time” as a single-copy property, just say “the leakage trace of p is a function of public inputs.” Section 3.2 notes that this approach runs into problems when what we call *compiler-resolved nondeterminism* is involved, but Section 3.3 presents a simple solution: get rid of the problematic nondeterminism by parameterizing the source-level semantics over an *oracle*.

It is natural to ask whether our approach can be made to work without getting rid of the source-level nondeterminism. In Section 4, we show that the answer is yes: we provide an alternative approach without determinizing the source-level semantics, and then we prove it equivalent (Theorem 4.3) to the semantics-determinization approach. In Section 5, we show again (in a different, subtly nonequivalent way) that the answer is yes, this time using the concept of a *predictor*, which is a dual concept to the oracles used in Section 3 and Section 4.

In Section 6, we shift our focus to compiler specifications. In Section 6.2, we propose a simple definition of “constant time-preserving” for a compiler: the compiler should admit *leakage-transformation functions*. However, Section 6.3 observes that requiring a compiler to admit leakage-transformation functions is insufficient when compiler-resolved nondeterminism is involved. As a fix, we propose two stronger compiler specifications: Definition 6.3 basically says that constant-time specifications in the style of Section 3 (or, equivalently, Section 4) are preserved by the compiler, while Definition 6.4 basically says that constant-time specifications in the style of Section 5 are preserved. It turns out that Definition 6.3 and Definition 6.4 are inequivalent in interesting ways; Section 6 concludes with a discussion of the tradeoffs between them.

We conclude with case studies. Section 7 describes our strategies for proving that the Bedrock2 compiler is constant time-preserving, and Section 8 describes our experience proving some Bedrock2 source programs correct and composing source-program theorems with the compiler theorem.

In total, the paper presents three different approaches to verifying constant time for source programs (and three corresponding compiler specifications): semantics determinization via oracles (Section 3.3); an equivalent approach, still with oracles but without semantics determinization (Section 4); and the approach with predictors (Section 5).

Our implementation is available as open source: compiler and source-program proofs in the semantics-determinization style appear in the main repository for Bedrock2². Proofs of the Bedrock2 compiler in all three styles are implemented in the artifact associated with this paper. Also, every fact presented in this paper as a theorem, lemma, or corollary is formalized in a separate repository³.

2 The Languages Used in This Paper

2.1 Background on Bedrock2 (Our Source Language)

Bedrock2 is an imperative language inspired by K&R C [17]. It has been used to create *end-to-end Coq proofs* about: a complete software-hardware IoT device controlling a lightbulb based on network inputs [13], a cryptographic server running on a microcontroller [15], and a selection of data structures [16]. As Bedrock2 has previously been used for “one Q.E.D.” proofs that span the

²<https://github.com/mit-plv/bedrock2>

³https://github.com/OwenConoly/semantics_relations

abstraction gap from relational specifications of applications down to hardware designs that run on FPGAs, we considered it a compelling baseline to extend in this work, to show that our new style of reasoning about timing channels is likely to apply to deep verified stacks, as well.

Syntax includes expressions and commands, with all variables storing machine words. Imperative state includes locals values ℓ and byte-granularity memory m . The state of the program also contains an input-output log t , preserved by compilation; and our addition, the leakage trace k .

Bedrock2 language features include functions, loops, conditionals, stack allocation, and I/O. There are no restrictions on pointer operations; pointers are simply machine words indexing into one flat memory space. They can be added, compared, printed, etc. Pointer-based data structures (e.g. linked lists) can be implemented in Bedrock2 as in C. Furthermore, Bedrock2's lower-level semantics simplify verification of memory-management code, as demonstrated by past work [16].

In this extension of Bedrock2 to handle constant time, we adapt the source-language semantics and compiler proof, but we leave for future work adaptation of processor proofs. As a result, our final theorems are about RISC-V machine code instead of hardware. However, machine-code semantics is our only trusted code; we do not trust the Bedrock2 source language or compiler.

It is also worth brief description of other complexities of the Bedrock2 compiler that we inherit. It works with not just the source language and RISC-V machine language but also one intermediate language flattened into three-address code. There are six phases: flattening, compilation of immediates, dead-code elimination, register allocation, spilling, and machine-code generation.

Two phases of the Bedrock2 compiler have different source and target languages. In our experience, verification of security properties has little interesting interaction with the difficulties of language heterogeneity; our techniques extend to the heterogeneous case without complication. So, for simplicity, our discussion will mostly stick to phases with matching source and target languages.

2.2 Bedrock2 Semantics

The canonical definition of Bedrock2 semantics is given as a set of weakest-precondition predicate-transformer rules that form an inductive definition of $p \Downarrow Q$. This form is closely related to traditional big-step semantics (for a well-defined terminating program, $\bigcap \{Q : p \Downarrow Q\}$ is the set of possible outcomes) and suitable for compiler verification by forward specification preservation [10].

Our semantics are the same as in the Bedrock2 paper [13], except we have added leakage traces k . A leakage trace includes two types of events: *leakage* and *compile-time nondeterminism* events. The CompNonDet events record how *compiler-resolved* nondeterministic events were resolved, while the Leak events, as in the previous high-level description of leakage trace, represent information that is leaked to an adversary. Concretely, we write a leakage trace as a list—for example,

[Leak x_1 , CompNonDet y_1 , Leak x_2 , Leak x_3 , CompNonDet y_2 , CompNonDet y_3 , ...],

where the x_i and y_i are machine words. In Bedrock2, CompNonDet events appear only in the stack-allocation rule (Section 3.3). However, our techniques apply equally well to other sources of compiler-resolved nondeterminism: nondeterministic evaluation order for expressions, PRNGs, etc.

The following big-step-omnisemantics [10, §2] rules illustrate our Bedrock2 semantics \Downarrow .

EVAL-STORE

$$\frac{(x, a) \in \ell \quad (a + n) \in \text{dom } m \quad (y, v) \in \ell \quad Q(m[(a + n) := v], \ell, t, k :: \text{Leak}(a + n))}{((x[n] = y), m, \ell, t, k) \Downarrow Q}$$

EVAL-INPUT

$$\frac{\forall n. Q(m, \ell[x := n], t :: \text{IN } n, k)}{((x = \text{input}()), m, \ell, t, k) \Downarrow Q}$$

EVAL-SEQ

$$\frac{(p_1, m, \ell, t, k) \Downarrow Q_1 \quad (\forall m', \ell', t', k'. Q_1(m', \ell', t', k') \implies (p_2, m', \ell', t', k') \Downarrow Q)}{((p_1; p_2), m, \ell, t, k) \Downarrow Q}$$

Observe that if a program that satisfies \Downarrow reads an input and then stores to an address that depends on that input, all possible store addresses must be in the domain of m , and the postcondition Q must accept all leakage traces (and memories) that may result. On the other hand, just reading an input and storing it to a fixed location $a[0]$ only requires Q to admit one leakage trace: $[\text{Leak } a]$.

We should also emphasize that Bedrock2 embodies a rather different compiler-verification approach than, e.g., CompCert’s [20]. CompCert handles complex nondeterminism through two main mechanisms: (1) a determinizing high-level memory model and (2) small-step semantics with a bestiary of simulation-proof techniques. Bedrock2 avoids the first complication by exposing a flat, machine code-style memory model at the source level. More interestingly, Bedrock2’s use of omnisemantics *enables all compiler-phase proofs to be performed using forward reasoning and big-step hypotheses*, even with nondeterministic target languages [10, §6]. Such a proof proceeds by simple induction over the big-step derivation for the source program, deriving the matching derivation for the target program. It was a nonobvious result that such a scheme works well for a language as expressive as Bedrock2, and with our work described in this paper, we add another surprising consequence: *proof of information-flow hyperproperties not just in big-step, forward-reasoning proof style, but also with no explicit consideration of multiple executions of source or target programs*.

Our work inherits a limitation from Bedrock2: the semantics of Bedrock2 only describe terminating (totally correct) programs, because omnisemantics has not yet been adapted to intentional nontermination (e.g., infinite reactivity). Thus, we consider nonterminating programs as out-of-scope. However, we expect our techniques to extend easily to any form of omnisemantics augmented to support nontermination. Further, even without omnisemantics support for nontermination, we should be able to lift our new results to infinitely reactive programs in the same way as did the original work with Bedrock2: by defining higher-level patterns like event loops [13].

2.3 RISC-V Semantics

For the target language of compiler-correctness proofs, we instrumented the Bedrock2 RISC-V semantics [8] with leakage traces. Like source-level Bedrock2 semantics, the RISC-V semantics is canonically interpreted as a weakest-precondition predicate transformer, which we modified also to log leakage. Our semantics additionally considers all instruction-fetch addresses to be leaked.

Following is an excerpt of our function computing the leakage of an instruction.

```
| Add _ _ _ => Return LeakAdd (* no arguments to LeakAdd, so inputs remain secret *)
| Lw _ rs1 _ => addr <- getReg rs1; Return (LeakLw addr)
| Blt rs1 rs2 _ => a <- getReg rs1; b <- getReg rs2; Return (LeakBlt (word.lts a b))
```

3 Specifying Constant Time Using Partially Determinized Semantics

In Section 1.2, with the username-password example, we illustrated a sufficient condition for recognizing constant-timeness of programs: namely, a program is *naive-constant-time* if its leakage is a function of public values (including both public initial state and public runtime input). In Section 3.1, we present examples of naive-constant-time specifications. However, in Section 3.2 we show that some intuitively constant-time programs involving compiler-resolved nondeterminism fail to satisfy naive constant time. Finally, in Section 3.3 we introduce the “compile-time-determinized” semantics $\Downarrow_{\mathcal{A}}$ as an expedient solution, allowing us to formulate a more permissive definition.

3.1 The Simple Case: No Leakage Dependence on Compiler-Resolved Nondeterminism

In this section, we illustrate specifications of naive constant time. The \Downarrow predicate we use in this section—and in the rest of this paper—is just like the one defined in Section 2.2, except the postcondition takes only two arguments (t, k) rather than four (m, ℓ, t, k) .

3.1.1 First Example: swap of Two Values Stored in Memory.

```
swap(int* pa, int* pb) { int tmp = *pa; *pa = *pb; *pb = tmp; }
```

Consider the addresses pa and pb to be public, while the values $*pa$ and $*pb$ stored at the addresses are private. Then we can specify naive-constant-timeness of `swap` as follows.

$$\exists f. \forall a_{\text{ptr}}, b_{\text{ptr}}, a, b. \forall m, \ell.$$

$$(\text{swap}(pa, pb), m[a_{\text{ptr}} := a, b_{\text{ptr}} := b], \ell[pa := a_{\text{ptr}}, pb := b_{\text{ptr}}], [], []) \Downarrow \{([], f(a_{\text{ptr}}, b_{\text{ptr}}))\}.$$

The final leakage is allowed to depend on the public values $a_{\text{ptr}}, b_{\text{ptr}}$ but not the private values a, b, m, ℓ . Thus, we interpret the specification above as saying that a, b, m, ℓ are not leaked.

The leakage of `swap` consists of a load at a_{ptr} , a load at b_{ptr} , a store at a_{ptr} , and a store at b_{ptr} . So, to prove the specification above, we define $f(a_{\text{ptr}}, b_{\text{ptr}}) := [\text{Leak } a_{\text{ptr}}, \text{Leak } b_{\text{ptr}}, \text{Leak } a_{\text{ptr}}, \text{Leak } b_{\text{ptr}}]$.

3.1.2 Second Example: login. (written in Pythonish syntax)

```
def login():
    username = input("enter your username")
    password = lookup_password(username)
    attempt = input("enter your password")
    print(if str_eq(password, attempt) then "hooray" else "wrong")
```

Consider the username to be public but the password to be private. Thus a constant-time program should have leakage independent of both the password and the password attempt. Here is a specification of the `login` function, expressing that the leakage only depends on the username u .

$$\exists f. \forall m, \ell. (\text{login}(), m, \ell, [], []) \Downarrow \{([(\text{IN } u, \text{IN } a, \text{OUT } r)], f(u)) : u, a, r \text{ are strings}\}.$$

However, the implementation of `login` does not actually meet this specification. The issue is that it branches on whether `password == attempt`, and we assume branches are leaked! So, it is unavoidable that the leakage depends on whether `password == attempt`.

The following subtler specification actually holds for the `login` function, assuming constant-time helper functions. We assume some pure function $L(m, u)$ that, given memory m and username u , returns the output of the `lookup_password` function on input u in m .

$$\exists f. \forall m, \ell. (\text{login}(), m, \ell, [], []) \Downarrow \{([(\text{IN } u, \text{IN } a, \text{OUT } r)], f(u, a == L(m, u))) : u, a, r \text{ are strings}\}.$$

It was easy to specify that, although the leakage may not depend on the attempt a or the password $L(m, u)$ in arbitrary ways, it is just allowed to depend on whether they are equal.

3.2 Problem: Naive Constant Time Forbids Compiler-Resolved Nondeterminism

```
stack_swap() { int s[2] = {0, 1}; swap(&s[0], &s[1]); }
```

Intuitively, `stack_swap()` should be a well-defined constant-time program with a no-op behavior. For `stack_swap` to be naive-constant-time, its leakage should be a function of public values. But its leakage depends on the address of the array `s`—which we model as being nondeterministic—so in fact the leakage is not a deterministic function of anything, let alone public values. We can still write a constant-time specification via an ad-hoc generalization of naive constant time. In addition to being allowed to depend on public initial state and I/O, we permit the leakage to depend arbitrarily on the allocation address recorded earlier in the leakage trace via `CompNonDet`:

$$\exists f. \forall m, \ell. (\text{stack_swap}(), m, \ell, [], []) \Downarrow \{([], [\text{CompNonDet } a] ++ f(a)) : a \text{ is a word}\}.$$

For this program it was clear what to say: the trace should begin with a nondeterministic event, and then the rest of it is allowed to depend on the outcome of that event (but not private values!) arbitrarily. However, more complicated programs could have `CompNonDet` events interleaved with

Leak events, with the length of the trace depending on the outcomes of earlier `CompNonDet`. So, general constant-time specifications appear to require elaborate postconditions detailing possible allocation patterns throughout execution. However, there is a simple alternative approach.

An alternative perspective on programs like `stack_swap` is that the difficulty arises because the source-language semantics are too nondeterministic. Our definition of \Downarrow says that the memory allocation is arbitrarily nondeterministic. But then the compiler is given too much flexibility; rather, it should somehow be encoded in the semantics that memory allocation is independent of, for instance, secrets stored in memory. To that end, the source-language semantics can instead say that memory allocation is some *unspecified deterministic function* of values that, in particular, do not include secrets stored in memory. We now expand upon this idea.

3.3 Solution: Constraining Inputs to Compiler-Resolved Nondeterministic Choices

If a source program’s behavior depends on some nondeterministic choice resolved by the compiler, it is natural to ask what the compiler’s choice is allowed in turn to depend on. Conveniently, it appears that the answer for each unspecified behavior we considered in `Bedrock2` is either “anything” or “very little.” For example, uninitialized *contents* of freshly allocated memory could depend on anything (e.g., recently deallocated memory). Thus, the programmer ought to avoid leaking these choices. In contrast, memory-allocation *addresses* are necessarily leaked by source programs through load and store commands. Thus, it is the compiler’s responsibility to ensure that its method of address selection does not reveal the program’s secrets. Similar considerations arise when outputting (rather than leaking) nondeterministically chosen values: revealing uninitialized memory contents could be compiled into an arbitrarily bad information leak, but printing a freshly allocated address is a common instrumentation technique—we will return to this aspect in [Section 3.3.2](#) and [Section 6.6](#).

The source-language semantics should specify what each compiler-implemented nondeterministic choice can depend on. Concretely, there are no restrictions on the data-flow dependencies of uninitialized data, and allocation addresses can only depend on information that the program has already leaked—for example, past control flow. Allowing memory-allocation addresses to depend arbitrarily on the leakage trace is a satisfying simplification: instead of separately keeping track of what could be leaked when the program reveals a choice made by compiler-generated code, the semantics consider everything that this choice could leak to be revealed eagerly.

More formally, our recommendation is to encode the compiler’s limited permission to make an implementation choice as an opaque (top-level existentially quantified) *oracle* function \mathcal{A} in the semantics. Each operational rule with limited nondeterminism calls \mathcal{A} with the leakage trace as an argument; the output of \mathcal{A} determines the outcome of the nondeterministic event. This flow corresponds to the perspective that, when the dependencies of a choice are limited to a subset of the program’s state, that choice is no longer nondeterministic.

We will write $\Downarrow_{\mathcal{A}}$ to denote a modified version of the `Bedrock2` semantics, where each stack-allocation address is picked using an opaque function \mathcal{A} of the leakage trace. The only difference between the two versions, \Downarrow and $\Downarrow_{\mathcal{A}}$, is how the address a of a fresh stack allocation is picked:

$$\frac{\forall a. \quad \forall m_{\text{new}}. \quad (\text{dom } m_{\text{new}} \cap \text{dom } m) = \emptyset \wedge \text{dom } m_{\text{new}} = [a, a + n) \implies (p, m \cup m_{\text{new}}, \ell[x := a], t, k :: \text{CompNonDet } a) \Downarrow \{(m', \ell', t', k') : Q(m' - m_{\text{new}}, \ell', t', k')\}}{((\text{stackalloc } n \text{ as } x; p), m, \ell, t, k) \Downarrow Q} \text{ alloc}$$

$$\frac{a = \mathcal{A}(k) \quad \forall m_{\text{new}}. \quad (\text{dom } m_{\text{new}} \cap \text{dom } m) = \emptyset \wedge \text{dom } m_{\text{new}} = [a, a + n) \implies (p, m \cup m_{\text{new}}, \ell[x := a], t, k :: \text{CompNonDet } a) \Downarrow_{\mathcal{A}} \{(m', \ell', t', k') : Q(m' - m_{\text{new}}, \ell', t', k')\}}{((\text{stackalloc } n \text{ as } x; p), m, \ell, t, k) \Downarrow_{\mathcal{A}} Q} \text{ alloc}_{\mathcal{A}}$$

We found this oracle-based encoding workable in both compiler and source-program proofs, but actually specifying the right semantics and understanding its implications were far from intuitive. For example, our first attempt did not log oracle calls in the leakage trace, so we had accidentally written semantics guaranteeing that any two consecutive (i.e., with no leakage events in-between) allocations have the same address. (Note that oracle calls are being logged with the `CompNonDet` event above, which serves the dual purpose of (1) logging the outcome of a nondeterministic event, which we will use in [Section 4](#), and (2) preventing the just-mentioned problem.)

Now that we have the $\Downarrow_{\mathcal{A}}$ predicate, it is straightforward to define constant time for arbitrary Bedrock2 programs. We say that a program is *constant-time* if it executes with a leakage trace that is a function of (1) the oracle \mathcal{A} and (2) public values.

3.3.1 Example: Now We Can Easily Write a Specification for `stack_swap`.

$$\exists f. \forall \mathcal{A}. \forall m, \ell. (\text{stack_swap}(), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{([], f(\mathcal{A}))\}.$$

Leakage is allowed to depend on \mathcal{A} and public values (there are none) but not on secrets in m, ℓ .

3.3.2 Example: Intentionally Outputting Compiler-Resolved Nondeterminism Is Safe per $\Downarrow_{\mathcal{A}}$.

```
stackalloc_and_print() { stackalloc 1 as x; print(x); }
```

Some security-related considerations are not captured by the leakage trace. For instance, suppose an adversary can read values that are printed; then, the printed values should not depend on secrets.

As the program above prints the *pointer* x (as opposed to uninitialized memory such as $*x$), we expect its output to be independent of secrets. We formalize this expectation as follows; the point is that the printed value may depend on \mathcal{A} , but it is independent of secrets in m and ℓ .

$$\exists f_{\text{io}}, f. \forall \mathcal{A}. \forall m, \ell. (\text{stackalloc_and_print}(), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{([\text{OUT } f_{\text{io}}(\mathcal{A})], f(\mathcal{A}))\}.$$

The `stackalloc_and_print` function is a toy example. For a more realistic example where we care about outputs not depending on secrets, consider a concurrent program. In Bedrock2, interaction with shared memory is modeled as I/O. Instead of printing (as in `stackalloc_and_print`), output might mean writing to shared memory. So, proving that outputs do not depend on secrets guarantees that the thread is not leaking secrets to other threads via memory.

4 Specifying Constant Time Using Oracles in the Postcondition

In [Section 3](#), we considered the problem of writing constant-time specifications in the context of compiler-resolved nondeterminism. We presented a simple solution: get rid of the problematic nondeterminism. This solution works well, but retaining nondeterminism can be convenient.

This section describes how to achieve the same results as in [Section 3](#) without resorting to the determinized semantics $\Downarrow_{\mathcal{A}}$. We want to find a more general solution than the ad-hoc approach used to write a specification of `stack_swap` in [Section 3.2](#). Finding a more general approach is important for two reasons. First, writing custom specifications for each function would be complicated and error-prone. Second, memory-allocation nondeterminism (unlike I/O) disappears after a program is compiled. Thus, we need to treat it uniformly so that we can prove a compiler theorem about it. (Roughly, the compiler theorem should say that if the source program is constant-time up to memory-allocation nondeterminism, then the target program is constant-time.)

4.1 How to Write Constant-Time Specifications with Nondeterministic Semantics

Suppose a program p is constant-time. Intuitively speaking, if we fix public values, then—as discussed in [Section 3](#)—the leakage of p should be a function of the oracle \mathcal{A} with which p executes. Our goal here, then, is to give a meaning to the phrase “the oracle \mathcal{A} with which p executes,” without resorting to the determinized $\Downarrow_{\mathcal{A}}$. This framing motivates the following definitions.

As a first step, we formalize how to say, in the context of \Downarrow rather than $\Downarrow_{\mathcal{A}}$, that “the nondeterminism was resolved according to \mathcal{A} ,” or “the leakage trace is compatible with \mathcal{A} .”

DEFINITION 4.1. *We say that a leakage trace k is compatible with an oracle \mathcal{A} , and we write $k \sim \mathcal{A}$, if for all k_1, x, k_2 such that $k = k_1 ++ [\text{CompNonDet } x] ++ k_2$, we have $\mathcal{A}(k_1) = x$.*

The point is that, in a postcondition of \Downarrow , the statement $k \sim \mathcal{A}$ holds if and only if \Downarrow happened to behave “as if it were $\Downarrow_{\mathcal{A}}$.” By only requiring a postcondition to hold when \Downarrow behaves like $\Downarrow_{\mathcal{A}}$, we can define a notion analogous to $\Downarrow_{\mathcal{A}}$ directly in terms of \Downarrow .

DEFINITION 4.2. *We say that $(p, m, \ell, [], []) \Downarrow_{\mathcal{A}}^{\star} Q$ if the postcondition Q holds whenever the program happens to execute compatibly with the oracle \mathcal{A} . Formally,*

$$(p, m, \ell, [], []) \Downarrow_{\mathcal{A}}^{\star} Q := (p, m, \ell, [], []) \Downarrow \{(t, k) : k \sim \mathcal{A} \Rightarrow Q(t, k)\}.$$

4.2 Equivalence of Intrinsically and Postcondition-Wise Oracle-Based Semantics

THEOREM 4.3. *For all p, m, ℓ, Q ,*

$$[\forall \mathcal{A}. (p, m, \ell, [], []) \Downarrow_{\mathcal{A}} Q(\mathcal{A})] \iff [\forall \mathcal{A}. (p, m, \ell, [], []) \Downarrow_{\mathcal{A}}^{\star} Q(\mathcal{A})].$$

So, in a strong sense, the $\Downarrow_{\mathcal{A}}^{\star}$ predicate is equivalent to the $\Downarrow_{\mathcal{A}}$ predicate. A surprising consequence is that \Downarrow can be used to express every specification—including constant-time specifications—that can be written with $\Downarrow_{\mathcal{A}}$. We do not need determinized semantics to define constant time.

Appendix B proves this theorem. The proof involves a few nonobvious technical devices but, we claim, does not depend on peculiarities of Bedrock2. More precisely: an analogous result should hold for any reasonable computer language with any nondeterministic construct being factored out into an oracle. In particular, the proof does not rely on e.g. finiteness of the machine-state type in Bedrock2. However—as with omnisemantics in general—it is not immediately obvious how to extend this result to fine-grained perpetually reactive behaviors.

4.3 Example: A Constant-Time Specification of `stack_swap` Using $\Downarrow_{\mathcal{A}}^{\star}$

Mirroring [Section 3.3.1](#), we can state that `stack_swap` is constant-time:

$$\exists f. \forall \mathcal{A}. \forall m, \ell. (\text{stack_swap}(), m, \ell, [], []) \Downarrow_{\mathcal{A}}^{\star} \{([], f(\mathcal{A}))\}.$$

It is a basic property of omnisemantics [10] that nonvacuous \forall quantifiers commute with \Downarrow . In particular, by unfolding $\Downarrow_{\mathcal{A}}^{\star}$, the above specification is equivalent to the following one:

$$\exists f. \forall m, \ell. (\text{stack_swap}(), m, \ell, [], []) \Downarrow \{([], k) : \forall \mathcal{A}. k \sim \mathcal{A} \Rightarrow k = f(\mathcal{A})\}.$$

5 Predictors: a Restricted Class of Functions Taking Oracles to Traces

Rephrasing the intuition we have been working from: a program p is constant-time if, after fixing public values, the trace of p is a function of the oracle \mathcal{A} with which p executes. Thus there should be a procedure \mathcal{P} for taking the oracle \mathcal{A} with which p executes and obtaining the trace of p .

Now, we consider what the procedure \mathcal{P} should look like. It should not need to query the oracle arbitrarily. For instance, \mathcal{P} should be able to generate a certain prefix of the trace—namely, the prefix before p takes any nondeterministic branches—without querying the oracle! In general, given any prefix of the trace of p , the procedure \mathcal{P} should be able to predict what the next event in the trace will be—except, when the next event is `CompNonDet`, then \mathcal{P} cannot know which way the branch goes, so it has to query the oracle.

We will now give a formal definition corresponding to the class of procedures \mathcal{P} we are talking about. As their function is to take a prefix of a trace and predict what comes next (modulo nondeterminism), we will call them *predictors*. We will say that \mathcal{P} *predicts* a trace k if, for every oracle \mathcal{A} compatible with k , the trace generated by running the procedure \mathcal{P} with the oracle \mathcal{A} will be k .

DEFINITION 5.1. A predictor is a function \mathcal{P} that takes in a trace and outputs one of the symbols CompNonDet , $\text{Leak } x$, or End . We say \mathcal{P} predicts a trace k , and we write $k \in \mathcal{P}$, if the following hold.

- $\forall k_1, k_2, x$, if $k = k_1 ++ [\text{Leak } x] ++ k_2$, then $\mathcal{P}(k_1) = \text{Leak } x$.
- $\forall k_1, k_2, x$, if $k = k_1 ++ [\text{CompNonDet } x] ++ k_2$, then $\mathcal{P}(k_1) = \text{CompNonDet}$.
- $\mathcal{P}(k) = \text{End}$.

DEFINITION 5.2. Let \mathcal{P} be any function that takes I/O traces to predictors. We say that $(p, m, \ell, [], [])$ executes with predictor \mathcal{P} if $(p, m, \ell, [], []) \Downarrow \{(t, k) : k \in \mathcal{P}(t)\}$.

We say that a program p is *predictor-constant-time* if it executes with a predictor depending only on public initial values and public runtime input. More precisely: let g be such that $g(t)$ comprises exactly the public information contained in the I/O trace t of p . Then p is *predictor-constant-time* if it executes with a predictor of the form $\mathcal{P} \circ g$, where \mathcal{P} depends only on public initial values.

Now, we show that this definition of predictors corresponds exactly to our intuitive thought of predictors as procedures that take oracles as input and return traces.

THEOREM 5.3. For any predictor \mathcal{P} , there exists a partial function $f_{\mathcal{P}}$, which takes oracles to leakage traces, such that the following holds.

$$\forall \mathcal{A}. \forall k. k \sim \mathcal{A} \implies [k \in \mathcal{P} \Leftrightarrow f_{\mathcal{P}}(\mathcal{A}) = \text{Some } k]$$

PROOF. The function $f_{\mathcal{P}}(\mathcal{A})$ is precisely the procedure outlined in the first two paragraphs of this section. We describe it as an imperative program. The program $f_{\mathcal{P}}(\mathcal{A})$ begins by setting the variable $k \leftarrow []$. Then, repeatedly: if the predicted next event $\mathcal{P}(k)$ is $\text{Leak } x$, then it sets $k \leftarrow k ++ [\text{Leak } x]$; else if $\mathcal{P}(k)$ is CompNonDet , then it sets $k \leftarrow k ++ [\text{CompNonDet } \mathcal{A}(k)]$; else if $\mathcal{P}(k)$ is End , then it returns $\text{Some } k$. (If this program loops, we say it returns None .) \square

5.1 Equivalence of Predictor Constant Time and (Oracle) Constant Time

Predictor constant time is a more structured and, a priori, stricter requirement than the notions of constant time developed in previous sections. Specifically, if a program is predictor-constant-time, then not only is its trace a function f of the oracle with which it executes, but the function f takes a certain simple form: in particular, it comes from some predictor via [Theorem 5.3](#).

COROLLARY 5.4. Let g be any function. (As before, we interpret $g(t)$ as being the public part of t .) For any \mathcal{P} , the following holds: for all p, m, ℓ ,

$$(p, m, \ell, [], []) \Downarrow \{(t, k) : k \in \mathcal{P} \circ g(t)\} \iff \forall \mathcal{A}. (p, m, \ell, [], []) \Downarrow_{\mathcal{A}}^* \{(t, k) : \text{Some } k = f_{\mathcal{P} \circ g(t)}(\mathcal{A})\}.$$

PROOF. Unfold $\Downarrow_{\mathcal{A}}^*$, and commute \mathcal{A} with \Downarrow to push $\forall \mathcal{A}$ into the righthand postcondition. Since every trace is compatible with some oracle, the postconditions are equivalent by [Theorem 5.3](#). \square

If a program p is predictor-constant-time, then it satisfies a specification of the form of the left-hand side of [Corollary 5.4](#). Therefore, it satisfies the specification on the right-hand side (which says that it is constant-time). Thus, predictor constant time implies constant time. The following theorem says the converse: if p is constant-time, then it is predictor-constant-time as well.

THEOREM 5.5. Let g and f be any functions. There exists some \mathcal{P} such that for all p, m, ℓ ,

$$[\forall \mathcal{A}. (p, m, \ell, [], []) \Downarrow_{\mathcal{A}}^* \{(t, f(g(t), \mathcal{A}))\}] \implies (p, m, \ell, [], []) \Downarrow \{(t, k) : k \in \mathcal{P} \circ g(t)\}.$$

PROOF. This is significantly harder than [Corollary 5.4](#), in that the postcondition on the left-hand side does not actually imply the right-hand postcondition. The proof must actually use the structure of the \Downarrow predicate. However, the main idea is just to invert the procedure from [Theorem 5.3](#). \square

5.2 Proving Programs Predictor-Constant-Time

For example, here is a specification saying that `stack_swap` is predictor-constant-time.

$$\exists \mathcal{P}. \forall m, \ell. (\text{stack_swap}(), m, \ell, [], []) \Downarrow \{([\], k) : k \in \mathcal{P}\}.$$

Looking at the definition of `stack_swap` (Section 3.2) and the definition of `swap` (Section 3.1.1), clearly the trace of `stack_swap` will always be of the form $[\text{CompNonDet } x; \text{Leak } x; \text{Leak } (x + 1); \text{Leak } x; \text{Leak } (x + 1)]$. Then it is clear how to define \mathcal{P} to prove the specification of `stack_swap`.

The full definition of \mathcal{P} would be awkward to write down here, but it is written in Appendix C.5. Predictors tend to be unwieldy and unilluminating in any notation that follows the definition directly, which makes them inconvenient for source-program proofs. On the other hand, we found them to be perfectly suitable for compiler proofs. Appendix C discusses predictors' drawbacks in more detail and presents a nicer, mostly equivalent way to represent them.

Using oracles to model compiler-resolved nondeterminism naturally leads to compositional verification using standard omnisemantics program-logic constructions [10, §5]. The same is true for predictors, but it requires a trick that we detail in Appendix D.

6 Constant-Time Compiler Specifications

Constant-time specifications can be subtle, and it would not be straightforward to write a compiler theorem directly saying that every possible source-level constant-time theorem implies a corresponding target-level constant-time theorem. So, our notion of compiler correctness will be intentionally more general than “compiling constant-time programs to constant-time programs.”

To avoid distractions, we discuss compiler passes $C : L \rightarrow L$ that work within a single language.

6.1 The Simple Case: Compiler Correctness for Fully Deterministic Languages

DEFINITION 6.1. C takes fixed-trace programs to fixed-trace programs if

$$\forall p. \exists \gamma_p. \forall k_H, m, \ell. (p, m, \ell, [], []) \Downarrow \{(t, k) : k = k_H\} \implies (C(p), m, \ell, [], []) \Downarrow \{(t, k) : k = \gamma_p(k_H)\}.$$

This way is the most direct to state that a compiler preserves deterministic constant time: high-level constant time is the hypothesis, and low-level constant time is the conclusion. It is precisely analogous to the specification (Theorem 5.1) used for deterministic languages in [4].

One can check that composing Definition 6.1 with our specification of the deterministic program swap from Section 3.1.1 yields a statement saying that $C(\text{swap}())$ is constant-time.

6.2 Compiler Correctness in the Presence of External Nondeterminism

Knowing that C takes fixed-trace programs to fixed-trace programs says nothing about its behavior on nondeterministic programs like `login()`. The issue is that Definition 6.1 effectively has an implicit hypothesis: the source program can only have one possible leakage trace! More precisely: since the leakage of `login()` depends on runtime input, no single k_H satisfying the hypothesis of Definition 6.1 exists.

To support programs that perform I/O, we generalize the compiler specification to say *unconditionally* that low-level leakage must be a deterministic function of high-level leakage.

DEFINITION 6.2. C admits leakage-transformation functions if

$$\forall p. \exists \gamma_p. \forall Q, m, \ell. (p, m, \ell, [], []) \Downarrow Q \implies (C(p), m, \ell, [], []) \Downarrow \{(t, \gamma_p(k_H)) : Q(t, k_H)\}.$$

If C admits a leakage-transformation function γ_p for $p := \text{login}$, we can show that the specification of `login` is preserved appropriately. Applying Definition 6.2 to the specification of `login` written in Section 3.1.2 yields the following low-level specification (where $p := \text{login}()$ for brevity).

$$\exists f. \forall m, \ell. (C(p), m, \ell, [], []) \Downarrow \{([\text{IN } u, \text{IN } a, \text{OUT } r], \gamma_p \circ f(u, a == L(m, u))) : u, a, r \text{ are strings}\}.$$

6.3 Compiler Correctness in the Presence of Compiler-Resolved Nondeterminism

As illustrated, requiring C to admit leakage-transformation functions guarantees that it preserves constant-time specifications, even when external nondeterminism is involved. The reason is that the postcondition Q can encode a relation between k_H and the I/O log t , and this relation remains meaningful on the low level because *the compiler preserves t* , so we have common references to the same nondeterministic events. In contrast, compiler-resolved nondeterministic events are not preserved by the compiler. Thus, requiring C to admit leakage-transformation functions is no longer sufficient in the presence of compiler-resolved nondeterminism.

For example, in the case of `stack_swap`, the high-level specification says that the leakage of `stack_swap()` is a function of the high-level oracle, and the desired low-level specification is that the leakage of $C(\text{stack_swap}())$ is a function of the low-level oracle. If C merely admits leakage-transformation functions, then the leakage of $C(\text{stack_swap}())$ depends only on the leakage of `stack_swap()`—but the leakage of `stack_swap()` depends in turn on the high-level oracle, so we cannot conclude that the leakage of $C(\text{stack_swap}())$ depends only on the low-level oracle.

To conclude that the leakage of $C(\text{stack_swap}())$ is a function of the low-level oracle, we must impose some additional constraint on the compiler. There are two options. (Note that the first is stricter; any compiler satisfying it automatically satisfies the second.)

Specification 1: The low-level leakage must only depend on the low-level oracle and the high-level leakage. In addition, the high-level oracle must depend only on the low-level oracle.

Specification 2: The function taking low-level oracle to low-level leakage must depend only on the function taking high-level oracle to high-level leakage.

If a compiler satisfies either Specification 1 or Specification 2, then it appropriately preserves the specification of `stack_swap`. However, Specifications 1 and 2 are subtly and significantly different. Specification 1 presents deterministic resolution of stack-allocation addresses as an absolute assumption—arbitrary conclusions about the source program’s behavior may depend on it.

For instance, suppose we want to show that our compiler appropriately preserves the specification of `stackalloc_and_print`. The conclusion follows easily for Specification 1: the specification of `stackalloc_and_print` (Section 3.3.2) says that the printed value is a function of the high-level oracle, and then Specification 1 says that the high-level oracle is a function of the low-level oracle. We conclude that the printed value is a function of the low-level oracle.

In contrast, Specification 2 does not imply that the compiler preserves the specification of `stackalloc_and_print`. Specification 2 only allows us to make conclusions about leakage, and the specification of `stackalloc_and_print` has nothing to do with leakage.

6.4 Formalizing Specification 1: Requiring Oracle-Transformation Functions

DEFINITION 6.3. C admits leakage-transformation and oracle-transformation functions if

$$\forall p. \exists \gamma_p, \mathcal{A}_p. \quad \forall Q, m, \ell. (\forall \mathcal{A}. (p, m, \ell, [], []) \Downarrow_{\mathcal{A}} Q(\mathcal{A})) \implies \\ (\forall \mathcal{A}. (C(p), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{(t, \gamma_p(k_H, \mathcal{A})) : Q(\mathcal{A}_p(\mathcal{A}))(t, k_H)\}).$$

The key points are that high-level oracle $\mathcal{A}_p(\mathcal{A})$ depends only on low-level oracle \mathcal{A} , and low-level leakage $\gamma_p(k_H, \mathcal{A})$ depends only on high-level leakage k_H and low-level oracle \mathcal{A} . Note: by Theorem 4.3, we could replace $\Downarrow_{\mathcal{A}}$ with $\Downarrow_{\mathcal{A}}^*$ in Definition 6.3, and the definition would be equivalent.

In the case of passes with different source and target languages, Definition 6.3 is still the correct notion; the low-level oracle just might look different. For instance, in the degenerate case where the low-level language is deterministic, we take the low-level oracle to have type `unit`. This pattern applies to the Bedrock2 compiler phase that first produces machine code—on the level of machine code, there is no stack-allocation nondeterminism and hence no oracle.

6.4.1 *Example: stack_swap.* Applying [Definition 6.3](#) to the specification of `stack_swap` from [Section 3.3.1](#) (and letting $p := \text{stack_swap}()$ for brevity), we get an appropriate low-level specification:

$$\exists f. \forall \mathcal{A}. \forall m, \ell. (C(p), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{([], \gamma_p(f \circ \mathcal{A}_p(\mathcal{A}), \mathcal{A}))\}.$$

6.4.2 *Example: stackalloc_and_print().* Applying [Definition 6.3](#) to the specification from [Section 3.3.2](#) of `stackalloc_and_print` (and letting $p := \text{stackalloc_and_print}()$), we get

$$\exists f_{\text{io}}, f. \forall \mathcal{A}. \forall m, \ell. (C(p), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{([\text{OUT } f_{\text{io}} \circ \mathcal{A}_p(\mathcal{A})], \gamma_p(f \circ \mathcal{A}_p(\mathcal{A}), \mathcal{A}))\}.$$

6.5 Formalizing Specification 2: Requiring Predictor-Transformation Functions

Without assuming that the high-level oracle is a function of the low-level oracle, it seems very tricky to prove Specification 2. The problem is that, a priori, the function taking high-level oracle to high-level leakage could be arbitrarily complicated. For instance, suppose the high-level leakage is $[]$ when $\mathcal{A}([\text{Leak } 0] * 1000) = 0$ and $[\text{Leak } 0]$ otherwise. It is unclear how to construct a corresponding function taking low-level oracle to low-level leakage. To make Specification 2 feasible to prove, we must restrict how the high-level leakage can depend on the high-level oracle.

There is a natural solution: the high-level leakage should only be allowed to depend on the high-level oracle via a predictor! That is, our specification should say that there exists a γ_p such that for any \mathcal{P} , if the high-level leakage is the function $f_{\mathcal{P}}$ of the high-level oracle, then the low-level leakage is the function $f_{\gamma_p(\mathcal{P})}$ of the low-level oracle.

DEFINITION 6.4. C admits predictor-transformation functions if

$$\forall p. \exists \gamma_p. \quad \forall Q, m, \ell. (p, m, \ell, [], []) \Downarrow Q \implies \\ (C(p), m, \ell, [], []) \Downarrow \{(t, k_L) : \exists k_H. Q(t, k_H) \wedge (\forall \mathcal{P}. k_H \in \mathcal{P} \implies k_L \in \gamma_p(\mathcal{P}))\}.$$

Just as [Definition 6.2](#) says that the low-level trace is a function of the high-level trace, [Definition 6.4](#) says that the low-level predictor is a function of the high-level predictor.

We emphasize that [Definition 6.4](#) differs from [Definition 6.3](#) along *two* dimensions. First, [Definition 6.4](#) is an instance of Specification 2 rather than Specification 1. This difference is unrelated to predictors; one could formalize Specification 2 without mentioning predictors. (As discussed, such a formalization seems infeasible to prove.) Second, [Definition 6.4](#) is (a priori) a restricted form of Specification 2: it only applies in the case where the high-level leakage depends on the high-level oracle via a predictor, not via an arbitrary function. However, by [Section 5.1](#), the high-level leakage depends on the high-level oracle via an arbitrary function *if and only if* it depends on the high-level oracle via a predictor. That is: since predictor constant time is equivalent to constant time, and [Definition 6.4](#) says that the compiler preserves predictor constant time, [Definition 6.4](#) also says that the compiler preserves constant time.

Just like [Definition 6.3](#), [Definition 6.4](#) generalizes straightforwardly to interlanguage passes. In the special case where there is no compiler-resolved nondeterminism on the low level, specifying a low-level predictor is equivalent to specifying a low-level leakage trace; so in this special case, [Definition 6.4](#) states that the low-level *leakage* is a function of the high-level predictor.

6.5.1 *Example: stack_swap.* Assume that C admits predictor-transformation functions. Then, applying [Definition 6.4](#) to the specification of `stack_swap` from [Section 5.2](#) (and letting $p := \text{stack_swap}()$ for brevity), we obtain the following low-level specification.

$$\exists \mathcal{P}. \forall m, \ell. (C(p), m, \ell, [], []) \Downarrow \{([], k_L) : \exists k_H. k_H \in \mathcal{P} \wedge (\forall \mathcal{P}_0. k_H \in \mathcal{P}_0 \implies k_L \in \gamma_p(\mathcal{P}_0))\}.$$

The above postcondition clearly implies the simpler postcondition $\{([], k_L) : k_L \in \gamma_p(\mathcal{P})\}$, which is exactly the statement that $C(p)$ is predictor-constant-time.

6.6 Extra Flexibility Afforded to the Programmer by Specification 1

As discussed in [Section 6.3](#), Specification 1 allows the source-level programmer to assume that a program will actually execute with some oracle, whereas Specification 2 makes no such promise! As we saw with `stackalloc_and_print`, Specification 2 only provides guarantees about leakage traces—on the other hand, Specification 1 allows us to guarantee that observable behavior of a program is independent of secrets, *even when that behavior is not captured in the leakage trace*.

6.7 Extra Flexibility Afforded to the Compiler by Specification 2

Specification 2 gives the compiler more freedom: compiler-resolved nondeterministic events *need not actually be a function of previous leakage trace*. Therefore, certain reordering optimizations that a compiler could perform violate Specification 1 but nevertheless satisfy Specification 2.

The problem with reordering optimizations is the following. If the compiler reorders some parts of a program, then some nondeterministic event N may end up being after some event E in the target program, whereas in the source program N came before E and hence the outcome of N was not allowed to depend on E . A compiler satisfying Specification 1, then, is not permitted to perform such reordering in the case that on the low level, the outcome of N would depend on E .

As a concrete example, Specification 1 does not permit a compiler to transform p into p' .

```
 $p := \{ \text{random as } x; z = *w; \text{print}(x); \}$ 
 $p' := \{ z = *w; \text{random as } x; \text{print}(x); \}$ 
```

Here, for simplicity, we replace the Bedrock2 construct `stackalloc as x` with the `random as x` construct, which works exactly the same way in that it binds x to a random number; the only difference is that it does no memory allocation. In Appendix E, we prove that a compiler transforming p into p' can satisfy Specification 2 ([Definition 6.4](#)) but not Specification 1 ([Definition 6.3](#)).

6.8 Variations on Specification 2

Specification 2 allows the compiler the extra freedom detailed in [Section 6.7](#). However, as discussed in [Section 6.6](#), Specification 2 only talks about functions taking oracles to *leakage traces*, and therefore Specification 2 is only helpful when you want to prove things about leakage traces.

If we wanted to make Specification 2 more useful, we could have it talk about functions taking oracles to other things besides leakage traces. An easy way to do this is just to modify our definition of leakage trace so that the “other things” that we are interested in appear in the leakage trace.

For example, to make Specification 2 preserve the specification of `stackalloc_and_print`, we could parameterize the semantics \Downarrow over a function that says, given an I/O event, what should be the corresponding events (if any) added to the leakage trace. By putting all the relevant information (and no more) in the leakage trace, we ensure that every theorem we care to prove about the low-level program is of the form “the low-level program executes with some particular predictor,” and hence Specification 2 is, from the programmer’s perspective, equally useful to Specification 1.

This approach sounds inconvenient but workable. Bedrock2 semantics are already parameterized over an I/O specification [13]. Augmenting this specification to say which inputs and outputs are private—and then writing a compiler specification in the style of Specification 2—would allow for the benefits of both Specification 2 and Specification 1. Thus, variations on [Definition 6.4](#) could allow for more useful compiler specifications than either of the specifications ([Definition 6.3](#) and [Definition 6.4](#)) we have presented here. We leave the details for future work.

7 Compiler Proof Techniques

In this section, we outline how we proved that the Bedrock2 compiler satisfies the specifications we have discussed. The compiler was already constant time-preserving; we did not need to modify it.

We completed proofs that the Bedrock2 compiler satisfies each of [Definition 6.3](#) and [Definition 6.4](#). In each case, we verified the usefulness of the specification by applying it to source-level constant-time theorems to yield assembly-level constant-time theorems. We proved [Definition 6.3](#) in both the form that it is stated as well as the equivalent form in terms of $\Downarrow_{\mathcal{A}}^*$ rather than $\Downarrow_{\mathcal{A}}$. The two proofs were similar; one proceeded by induction on \Downarrow and the other by induction on $\Downarrow_{\mathcal{A}}$.

In [Section 7.2](#), we show how to construct leakage-, predictor-, and oracle-transformation functions. We wrote these directly in Gallina (Coq's dependently typed language) using well-founded recursion. Then, we proved the correctness of the transformation functions (e.g. that γ_p and \mathcal{A}_p satisfy [Definition 6.3](#)) by induction on $(p, m, \ell, t, k) \Downarrow Q$, applying the introduction rules of \Downarrow to symbolically execute each statement generated by the compiler pass C . Detailed examples and explanations of such proofs for Bedrock2 and lambda calculi can be found in Charguéraud et al. [[10](#), §6].

We emphasize that omnisemantics makes compiler proofs easy, and our leakage-related additions did not require any changes to the structure of the Bedrock2 compiler proofs.

7.1 Simple Case: Leakage-Preserving Optimizations

Some optimization passes are *leakage-preserving*: the target program always has the same leakage as the source program. This strong property straightforwardly implies whichever compiler specification we want to prove; the leakage-/predictor-/oracle-transformation functions are identities.

7.2 Constructing Transformation Functions

Leakage-Transformation Functions γ_p . The idea is that knowing the high-level leakage (especially, knowing which way branches go) allows us to simulate the execution of the source program, even though we know nothing about the initial state of the source program. Then, since we know the function C taking the source program to the target program, knowing source-level control flow yields knowledge of target-level control flow and in turn target-level leakage.

Concretely, a leakage-transformation function walks through the execution of the source program, guided by the source leakage trace telling it which branches to take. For each statement executed in the source program, it looks at the leakage of that statement and computes the corresponding target-level leakage. For example, the address of a source-language load appears in the source-language leakage trace and is reinserted into the target-language leakage trace if the compiled code also contains a corresponding load. On the other hand, addresses of target-level loads to stack slots only depend on the stack-frame address, which is known from control flow.

Predictor-Transformation Functions γ_p . To construct a predictor-transformation function, we assume given a high-level predictor \mathcal{P}_H , with which the source program executes; and a low-level leakage trace k_L , which is a prefix of the target-program leakage trace. The requirement is to predict what comes after k_L . To do so, the predictor-transformation function walks the source program very similarly to a leakage-transformation function, keeping track of the high-level leakage trace as it goes and using \mathcal{P}_H to see what comes next. When it has simulated the target program far enough to get to the end of k_L , it checks what event comes next based on control flow of the target program. If the next event is compiler-resolved nondeterminism, `CompNonDet` is returned, otherwise the event itself (possibly by querying \mathcal{P}_H to get the corresponding high-level event).

Oracle-Transformation Functions \mathcal{A}_p . To construct an oracle-transformation function, we assume given a low-level oracle \mathcal{A}_L , with which the target program executes; and a high-level leakage trace k_H , which is a prefix of the source-program leakage trace. The requirement is to return the outcome of the high-level compiler-resolved nondeterminism that comes after k_H . To do so, an oracle-transformation function walks through the source and target programs, guided by k_H and \mathcal{A}_L . When it reaches the end of k_H , it looks at the next instance of compiler-resolved nondeterminism in

the source program and determines the outcome of this choice by consulting the current low-level state (e.g., the value of a stack pointer or frame pointer) and the low-level oracle \mathcal{A}_L .

7.3 Transformation-Function Examples

7.3.1 Introducing Nondeterminism in the Spilling Phase. Here we illustrate how to construct the function γ_p of [Definition 6.3](#) when C is the spilling phase of our compiler. The spilling phase consists of a pair of functions `spill_stmt` and `spill_fun`. The former compiles a statement. The latter takes as input a function—that is, a list of argument names, a list of return-value names, and a function body—and returns the body of the compiled function.

```
Definition spill_fun argnames resnames body (words_needed : Z) : stmt :=
  stackalloc (bytes_per_word * words_needed) as fp;
  set_vars_to_reg_range argnames a0;
  spill_stmt body;
  set_reg_range_to_vars a0 resnames.
```

We have corresponding leakage-transformation functions `leak_stmt` and `leak_fun`. The leakage of a function body will depend on the value of the frame pointer `fp`. So, the `leak_stmt` function needs to take the following inputs: the low-level oracle \mathcal{A}_L , the high-level program sH and its leakage kH , the current value `fpval` of the dedicated frame-pointer register `fp`, and the low-level leakage that has accumulated so far `kL_so_far`. It will return the total low-level leakage that will have accumulated after sH finishes executing. In Coq:

```
Definition leak_stmt (AL : leakage -> word) (sH : stmt) (kH : leakage)
  (fpval : word) (kL_so_far : leakage) : leakage := ...
```

We consider how to implement `leak_fun` given `leak_stmt`. The specification of `leak_fun` is as follows: given a high-level function (`argnames`, `resnames`, `body`), a low-level oracle \mathcal{A}_L , the high-level leakage kH , and the low-level leakage-so-far `kL_so_far`, return the total leakage accumulated after the body of the spilled function finishes executing. It is implemented as follows.

```
Definition leak_fun argnames resnames body (AL : leakage -> word) (kH : leakage)
  (kL_so_far : leakage) : leakage :=
  let fpval := AL kL_so_far in
  let kL' := [CompNonDet fpval] ++ leak_set_vars_to_reg_range fpval argnames in
  leak_stmt AL body kH fpval (kL_so_far ++ kL')
  ++ leak_set_reg_range_to_vars fpval resnames.
```

To understand what is going on in `leak_fun`, compare it to `spill_fun`. The function `leak_fun` begins by querying the low-level oracle to see what value is put in the register `fp`. Then, it computes the leakage of the `stackalloc` call (which is `CompNonDet fpval`) and the leakage of the `set_vars_to_reg_range` operation. It passes the results to `leak_stmt`, which returns the leakage that has accumulated after `spill_stmt body` executes. Then it finishes by appending the leakage of the `set_reg_range_to_vars` operation.

7.3.2 Resolving Nondeterminism in the FlatToRiscv Phase. Here we illustrate how to construct the functions \mathcal{A}_p and γ_p of [Definition 6.3](#) when C is the FlatToRiscv phase of our compiler. This phase consists of a function `compile_stmt`, which takes as input `mypos`, the relative position of the output code relative to a base position; `stackoffset`, a value such that `stackoffset + sp_val` is the highest used stack address (where `sp_val` is the current value in the stack-pointer register `sp`); and a statement sH to be compiled. It outputs a list of assembly instructions.

```
Fixpoint compile_stmt(mypos: Z)(stackoffset: Z)(sH: stmt): list Instruction :=
  match sH with
```

```
| stackalloc n as x; body =>
  [Addi x sp (stackoffset-n)] ++ compile_stmt (mypos + 4) (stackoffset-n) body
... (* other cases omitted *) end.
```

We have a corresponding leakage-transformation function `leak_stmt`, which takes the following inputs: the high-level program `sH` with leakage trace `kH`, the values `mypos` and `stackoffset` (which have the same meanings as before), and the current value `sp_val` of the stack-pointer register `sp`. It outputs the leakage of the compiled program. Note that there is no compiler-resolved nondeterminism on the low level (i.e., the low-level oracle has type `unit`), so `leak_stmt` (unlike the `leak_stmt` of [Section 7.3.1](#)!) does not need to take a low-level oracle as input.

```
Fixpoint leak_stmt (sH : stmt) (kH : leakage) (mypos stackoffset sp_val : word)
: list LeakageEvent :=
match sH with
| stackalloc n as x; body =>
  match kH with
  | CompNonDet _ :: kH' =>
    [ LeakAddi ] ++ leak_stmt body kH' (mypos + 4) (stackoffset - n) sp_val
  | _ => (*should be impossible! kH is the leakage of an execution of sH*)
  end ... (*other cases omitted*) end.
```

To understand `leak_stmt`, compare it to `compile_stmt`. The function `leak_stmt` begins by breaking the high-level leakage `kH` into two pieces: the leakage of the line `stackalloc n as x`, which is `CompNonDet _`; and the leakage of the body that comes after, which is `kH'`.

We also have an `oracle_stmt` function, which takes the same inputs as `leak_stmt`; the only difference is that with `leak_stmt` the input leakage `kH` was expected to be the leakage of the whole high-level program `sH`, whereas with `oracle_stmt`, it is only expected to be some prefix of the leakage of `sH`. Then `oracle_stmt` returns the output of the high-level oracle when given `kH`.

```
Fixpoint oracle_stmt sH kH mypos stackoffset sp_val : word :=
match sH with
| stackalloc n as x; body =>
  match kH with
  | CompNonDet _ :: kH' => oracle_stmt body kH' (mypos+4) (stackoffset-n) sp_val
  | [] => sp_val + (stackoffset - n)
  | _ => (*this case should be impossible!*)
  end ... (*other cases omitted*) end.
```

If `kH` is the empty leakage trace, then `oracle_stmt` has finished simulating the source and target programs, and it should simply output the value bound to `x` by the high-level `stackalloc` call. Looking at `compile_stmt`, we see that the value bound to `x` is `sp_val + (stackoffset - n)`, so it is what `oracle_stmt` returns. On the other hand, if `kH` is not the empty leakage trace, then `oracle_stmt` is not yet done simulating the source and target programs, so it passes the appropriate values to the recursive call and continues by simulating `body`.

7.4 A Summary of Our Compiler Proof Effort

The whole project, to the point of submitting this paper, took about 12 months. Most of the time was spent trying to find the right specification of the compiler to handle stack allocation and trying to find the right strategy for writing the compiler proof. The adaptation of the Bedrock2 compiler proof was done by one person, who spent 18 hours per week on the project over 38 weeks.

Table 1. Lines of code written to prove the Bedrock2 compiler satisfies Definition 6.3

Pass	Base (proof lines before)	Proof lines added, deleted	Transf.-func. lines
FlattenExpr	956	+ 48, - 44	0
RegAlloc	1353	+ 10, - 8	0
UseImmediate	181	+ 18, - 26	0
DeadCodeElim	386	+ 140, - 160	222
Spilling	2034	+ 292, - 236	300
FlatToRiscv	4793	+ 426, - 131	484
composition	1499	+ 416, - 164	0

In total, we experimented with three approaches to verifying that the Bedrock2 compiler is constant time-preserving. We proved, by induction on $\Downarrow_{\mathcal{A}}$, that it satisfies the version of Definition 6.3 written in terms of $\Downarrow_{\mathcal{A}}$. Separately, we proved, by induction on \Downarrow , that it satisfies the version of Definition 6.3 written in terms of $\Downarrow_{\mathcal{A}}^*$. Finally, we proved, by induction on \Downarrow , that it satisfies Definition 6.4. All three approaches required a similar amount of effort.

Table 1 summarizes code written per compiler pass to prove Definition 6.3 by induction on $\Downarrow_{\mathcal{A}}$. Since leakage- and oracle-transformation functions are structured similarly, we did not write them separately, instead writing one function returning a tuple (leakage, oracle output).

The proof work consisted mainly of small edits to existing proof scripts. Note that most of the code is in the bodies of the leakage-/oracle-transformation functions. Adapting the FlattenExpr, RegAlloc, and UseImmediate phases was little work, as they are leakage-preserving (Section 7.1).

We also encountered a lucky natural experiment to let us measure directly effort to adapt a new phase: the DeadCodeElim phase was added to the Bedrock2 compiler during our project, and we adapted that phase very recently. Having the other phase proofs to reference, adapting the DeadCodeElim phase took only one day (about 5 hours).

8 Source-Program Case Studies

The case studies described in this section followed the semantics-determinization approach with $\Downarrow_{\mathcal{A}}$ (Section 3.3). All case studies are included in our Coq implementation.

8.1 Common Compiler Gotchas for Constant-Time Cryptography

Division by Constants. Compilers choosing non-constant-time implementation strategies for straightforward source-level constructs has led to numerous security issues in cryptographic software. Two recent and straightforward examples are the *lower* compiler optimization generating non-constant-time machine instructions for division and modulo by a constant in the NIST-PQC-competition algorithms Kyber and HQC, leading to exploitable vulnerabilities [7, 23]. Our semantics also afford the compiler this flexibility: division and modulo leak their arguments. However, the recommended fix (choosing the right implementation strategy at the source level) can be proven constant-time using our semantics. Specifically, we transcribed the fixed `poly_tomsg` to Bedrock2 and proved that its leakage trace depends only on the memory addresses of the input and output, not the scalar values in these arrays.

Constant-Time Memory Comparison. In most languages, unexpectedly clever compiler optimizations can detect functional behavior of constant-time idioms and replace them with (average-case faster) versions that leak information. A recurring example of this phenomenon, and a staple constant-time function found in every good cryptography library’s “subtle” section, is comparing two buffers for equality without leaking anything else about their contents. (Naively returning “not

equal” on the first mismatch would leak the length of the common prefix, allowing for incremental guessing.) We proved that the `memequal` function added to the Bedrock2 standard library as a part of the Bedrock2-Fiat-Crypto integration [15] indeed does not leak its in-memory inputs:

```
Definition memequal := func! (x,y,n) ~> r { r = $0;
  while n { r = r | (load1(x) ^ load1(y)); x = x + $1; y = y + $1; n = n - $1 };
  r = (r == $0) }.
```

We proved a source-level specification of `memequal`. Then we compiled it and applied a compiler theorem (following Definition 6.3), obtaining the following (abridged) assembly-level specification.

```
Lemma memequal_ct : forall x y n pos stack_pointer ret_addr,
  exists finalK, forall (xs ys : list word) stack_space initialMachine,
    (*hypothesis saying xs and ys are length-n arrays at addresses x and y*) ->
    (*hypothesis saying stack_space is big enough for memequal to run*) ->
    initialMachine.(getPc) = pos /\ initialMachine.(getLeakage) = [] ->
    map.get initialMachine.(getRegs) ra = Some ret_addr ->
    arg_regs_contain initialMachine.(getRegs) [x; y; n] ->
    (*hypothesis saying the instructions of memequal are at address pos*) ->
    runsTo initialMachine (fun finalMachine : RiscvMachine =>
      finalMachine.(getPc) = ret_addr /\ finalMachine.(getLeakage) = finalK).
```

Note that the leakage `finalK` is independent of, for instance, the content `xs` and `ys` of the arrays at locations `x` and `y`. In fact, `finalK` depends only on the arguments `x`, `y`, `n`, the address `pos` of the compiled `memequal` function, the stack pointer `stack_pointer`, and the return address `ret_addr` (leaked when the machine jumps back there).

Asking Coq to Print Assumptions `memequal_ct` yields only propositional and functional extensionality. The only other trusted code here (other than Coq) is our leakage-augmented RISC-V semantics, encapsulated in the `runsTo` predicate visible in the code above.

8.2 Password-Based Login

Our next software case study is a password prompt inspired by `agetty` used for Linux console login. The program reads a password character-by-character and compares it against a reference value, returning whether the password was correct, without leaking either operand. While the conceptual task of this example is just to call `memequal`, including some application context helps illustrate the different types of nondeterminism supported by our updated Bedrock2 compiler.

First, the program stack-allocates a temporary buffer for the user input. Per $\Downarrow_{\mathcal{A}}$, the allocated address is independent of the password stored in program memory. This fine point is important, as `getline` writing the entered password to that buffer leaks the address of the buffer.

Second, `getline` reads the user input by calling `getchar` until a newline is entered. As a result, the program branches on every character of the entered password compared with a newline! Nevertheless, we prove that the leakage of `getline` only depends on the *length* of the input `bs`, as well as the length `n` and address `dst` of the destination memory buffer. More specifically: for any \mathcal{A} ,

$$\exists f. \forall m, \ell. (\text{getline}(\text{dst}, n), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{(\text{getline_io}(\text{bs}), f(|\text{bs}|, \text{dst}, n)) \mid \text{passwords } \text{bs}\}.$$

Note that the distinction between values of `bs` (which are secret) and their lengths is program-specific, not understood by the compiler. Nevertheless, preserving leakage-free execution during compilation is a genuine obligation: replacing each character-newline comparison with a 256-entry jump table as a misguided optimization would leak every character’s value, not just the length.

Finally, the login program calls `memequal` to compare the entered and actual passwords. The overall postcondition establishes that the return value is 1 iff the password from `getline` is correct, but the leakage trace can be predicted based on memory addresses and lengths alone.

To further exercise corner cases of the semantics, we also proved a version where `memequal` is used to compare the correct password against the entire buffer passed to `getline`, not just the initialized part. In Bedrock2, unlike ISO C, branching on uninitialized values is defined behavior, so this program is legal, and the postcondition of `memequal` still guarantees that the program does not leak any information about the password, even though the uninitialized bytes may depend on it.

```
Definition password_checker := func! (password) ~> ret {
  stackalloc 8 as x; (* password is 8 characters *) unpack! n = getline(x, $8);
  unpack! ok = memequal(x, password, $8); ret = (n == $8) & ok }.
```

Implementing and proving this example (including `getline`) took less than one workday, with less than an hour spent on leakage proofs.

8.3 Output Without Leaking: Semiprime Generator

We opted not to consider program inputs or outputs as leaked, since such direct information flows can be ruled out in the original Bedrock2 program logic. This choice also gives us a chance to model cryptographic code in a useful way, where we assume an adversary does not have the computational power to e.g. factor a large secret key that has been output. As an example of this strategy, we proved that the following program that reads two prime numbers as runtime input and outputs their product does not leak the factors:

```
Definition semiprime := func! () ~> (p, q) {
  p = getprime(); q = getprime(); n = p * q; output(n) }.
```

Specifically, we established the following specification: for any \mathcal{A} ,

$$\exists f. \forall m, \ell. (\text{semiprime}(), m, \ell, [], []) \Downarrow_{\mathcal{A}} \{([getprime(p), getprime(q), output(pq)], f())\}.$$

8.4 Software Constant-Time-Verification Effort

Focusing on semantics and compiler verification, we have not invested any effort to optimize or streamline the software proof flow. Nevertheless, for all programs we have proven constant-time using any of our semantics, the time and code required to prove the actual constant-time property is a small fraction (around 10%) of proving memory safety and defined behavior. In particular, Coq's built-in proof-context tracking can automatically prove that an expression does not depend on a particular variable. Most examples took less than one working day to implement, specify, and prove, except for Kyber message decoding, which took less than one workweek.

9 Related Work

A variety of projects have applied formal verification to different types of compilers for general-purpose languages, most notably CompCert [20] and CakeML [19]. Some projects have been more specialized to the domain of cryptography, where the idea of constant time originated. Fiat Cryptography [14] produces (with a Coq-verified compiler) performance-critical inner loops that are straightline code without explicit memory access and thus trivially verified as constant time. The libraries HACL* [28] and EverCrypt [22] based on the F* language [25] tackle a broader range of cryptographic functionality, where constant time must be established explicitly. There are no associated mechanized proofs about compilation, not even for functional correctness.

Barthe et al. [6] initiated mechanized proof that compilers preserves constant time, first for a simple compiler inspired by Jasmin [2]. We already discussed work [4] applying similar techniques to most phases of CompCert. In general, this past work did not apply to nondeterministic programs.

Barthe et al. [5] introduce the idea of *structured leakage*, where leakage traces are accumulated in purpose-specific data structures. They handle allocation of stack memory, one of our main running examples. However—as with CompCert—their source language provides no way to observe memory-allocation addresses, sidestepping some of the central challenges that motivate our work. In contrast, our approach permits proving a program constant-time even if it branches on the stack pointer. We also handle other kinds of nondeterminism, most interestingly input-output.

We are not aware of prior work involving specification of constant-time properties either (1) as single-copy properties or (2) using anything analogous to our “predictors.” Also, most prior work on verification of compiler security properties—including all works cited in this section—uses small-step semantics and thus (unlike our approach) supports nontermination without difficulty.

We did not consider fine-grained security policies, merely tagging each value “private” or “public.” Broberg et al. [9] overview a broader class of security policies.

Constant Time With Nondeterminism. Sison and Murray [24] prove for a realistic concurrent program that threads do not leak to other threads via shared memory or timing leaks. Their work significantly differs from ours in that they work with security properties as hyperproperties rather than single-copy properties. A significant part of their contribution is a method for proving a certain security hyperproperty by considering only two executions at a time (one source, one target) rather than four at a time (two source, two target). We avoid relating imperative executions at all.

Muller and Chong [21] also handle security properties in the presence of concurrency, but they focus only on static analysis, not considering compiler specifications or proofs. To do so, they parameterize their semantics over *refiners* to make it deterministic, just like we parameterize our semantics over oracles in Section 3.3. We used oracles to model stack allocation, and they used refiners to model scheduling (for concurrency), but it is the same idea.

Speculation-Aware Security Properties. In general, cryptographic constant time is an insufficient security condition in the face of hardware optimizations like speculation, as demonstrated with Spectre attacks [18]. Tools based on programming languages and compilers have been suggested that help regain timing security in the face of speculation. One example is Blade [27], which employs a type system to track information flows and ensure sufficient placement of relatively expensive source-level mitigations.

Recently, both Arranz Olmos et al. [3] and van der Wall and Meyer [26] have demonstrated how to verify compilers against speculation-aware semantics. Both consider an adversary’s actions to be a source of nondeterminism, modeled by parameterizing the semantics over *directives* (analogous to oracles and refiners), which “model the ability of an adversary to influence program execution” [3]. These speculation-aware notions of timing security may also benefit from generalization to different kinds of nondeterminism, as we have explored for traditional constant time.

10 Conclusion

Characterizing timing security of interactive programs requires going beyond cryptographic constant time, even as the ideas behind that established notion remain relevant. We introduced a series of approaches to specifying security of interactive programs, proving appropriate relations between approaches and showing how compiler verification can be adapted to show preservation of specifications in these different styles. Ideas for building on our results include connecting security proofs to include end-to-end results with respect to specific processors, bringing in some notion of computational complexity to rule out unrealistic adversaries, and extending to proof of new phases that ingest security specifications and use them to drive additional optimizations.

Artifact-Availability Statement

An artifact [11] for this paper was evaluated and is freely available.

Acknowledgments

This research was supported by the National Science Foundation under grant CCF-2130671. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Nadhem J. Al Fardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, USA, 526–540. <https://doi.org/10.1109/SP.2013.42>
- [2] 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. <https://doi.org/10.1145/3133956.3134078>
- [3] 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. <https://doi.org/10.1145/3704880>
- [4] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL, Article 7 (Dec 2019), 30 pages. <https://doi.org/10.1145/3371075>
- [5] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, Republic of Korea) (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 462–476. <https://doi.org/10.1145/3460120.3484761>
- [6] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time”. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 328–343. <https://doi.org/10.1109/CSF.2018.00031>
- [7] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. 2024. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. *Cryptology ePrint Archive*, Paper 2024/1049. <https://eprint.iacr.org/2024/1049>
- [8] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. 2023. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 192 (aug 2023), 17 pages. <https://doi.org/10.1145/3607833>
- [9] Niklas Broberg, Bart van Delft, and David Sands. 2015. The Anatomy and Facets of Dynamic Policies. In *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium (CSF '15)*. IEEE Computer Society, USA, 122–136. <https://doi.org/10.1109/CSF.2015.16>
- [10] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 5 (mar 2023), 43 pages. <https://doi.org/10.1145/3579834>
- [11] Owen Conoly, Andres Erbsen, and Adam Chlipala. 2025. *Machine-Checked Proof Artifact for Integrated Proofs of Cryptographic Constant Time for Nondeterministic Programs and Compilers*. <https://doi.org/10.5281/zenodo.15043688>
- [12] Owen Conoly, Andres Erbsen, and Adam Chlipala. 2025. Smooth, Integrated Proofs of Cryptographic Constant Time for Nondeterministic Programs and Compilers. *arXiv:2504.15550 [cs.PL]* <https://arxiv.org/abs/2504.15550>
- [13] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 604–619. <https://doi.org/10.1145/3453483.3454065>
- [14] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. *2019 IEEE Symposium on Security and Privacy (SP)* 54, 1 (May 2019), 1202–1219. <https://doi.org/10.1109/sp.2019.00005>
- [15] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. *Proc. ACM Program. Lang.* 8, PLDI, Article 216 (jun 2024), 26 pages. <https://doi.org/10.1145/3656446>

- [16] Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Live Verification in an Interactive Proof Assistant. *Proc. ACM Program. Lang.* 8, PLDI, Article 209 (June 2024), 24 pages. <https://doi.org/10.1145/3656439>
- [17] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2 ed.). Prentice Hall.
- [18] 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. 2020. Spectre attacks: Exploiting Speculative Execution. *Commun. ACM* 63, 7 (18 June 2020), 93–101. <https://doi.org/10.1145/3399742>
- [19] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *SIGPLAN Not.* 49, 1 (Jan. 2014), 179–191. <https://doi.org/10.1145/2578855.2535841>
- [20] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (Nov. 2009), 363–446. <https://doi.org/10.1007/s10817-009-9155-4>
- [21] Stefan Muller and Stephen Chong. 2012. Towards a practical secure concurrent language. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) (OOPSLA '12). Association for Computing Machinery, New York, NY, USA, 57–74. <https://doi.org/10.1145/2384616.2384621>
- [22] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramanananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. 983–1002. <https://doi.org/10.1109/SP40000.2020.00114>
- [23] Robin Leander Schröder, Stefan Gast, and Qian Guo. 2024. Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks. Cryptology ePrint Archive, Paper 2024/299. <https://eprint.iacr.org/2024/299>
- [24] Robert Sison and Toby Murray. 2019. Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security. In *International Conference on Interactive Theorem Proving* (2019–9–6), Vol. 141. Schloss Dagstuhl, Portland, USA, 27:1–27:19. <https://doi.org/10.4230/LIPIcs.ITP.2019.27>
- [25] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) (POPL '16). Association for Computing Machinery, New York, NY, USA, 256–270. <https://doi.org/10.1145/2837614.2837655>
- [26] 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. <https://doi.org/10.1145/3704887>
- [27] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with Blade. *Proc. ACM Program. Lang.* 5, POPL, Article 49 (jan 2021), 30 pages. <https://doi.org/10.1145/3434330>
- [28] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACl*: A Verified Modern Cryptographic Library. 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, 1789–1806. <https://doi.org/10.1145/3133956.3134043>

Received 2024-11-14; accepted 2025-03-06