Certified Compilation based on Gödel Numbers

GUILHERME DE OLIVEIRA SILVA, Cadence, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

In his 1984 Turing Award lecture, Ken Thompson showed that a compiler could be maliciously altered to insert backdoors into programs it compiles and perpetuate this behavior by modifying any compiler it subsequently builds. Thompson's hack has been reproduced in real-world systems for demonstration purposes. Several countermeasures have been proposed to defend against Thompson-style backdoors, including the well-known *Diverse Double-Compiling* (DDC) technique, as well as methods like translation validation and CompCert-style compilation. However, these approaches ultimately circle back to the fundamental question: "*How can we trust the compiler used to compile the tools we rely on?*" In this paper, we introduce a novel approach to generating certificates to guarantee that a binary image faithfully represents the source code. These certificates ensure that the binary contains all and only the statements from the source code, preserves their order, and maintains equivalent def-use dependencies. The certificate is represented as an integer derivable from both the source code and the binary using a concise set of derivation rules, each applied in constant time. To demonstrate the practicality of our method, we present CHARON, a compiler designed to handle a subset of C expressive enough to compile FACT, the Flexible and Constant Time cryptographic programming language.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Compilers**.

Additional Key Words and Phrases: Compilation, Certification, Gödel Numbering

1 Introduction

In his Turing Award lecture, "*Reflections on Trusting Trust*" [Thompson 1984], Ken Thompson introduced an influential concept in cybersecurity: he described an attack in which a compiler is modified to insert a backdoor into the UNIX login command. The brilliance of this attack lies in its ability to self-propagate: the compromised compiler not only injects the backdoor but also recognizes and reinserts it into new compiler versions, even if the malicious code is removed from the source. The concrete implementation of the attach is relatively simple [Cox 2023], something noticeable, given how hard is to prevent it. Thompson's work highlights two critical truths: software tools can be compromised in ways that evade traditional inspection, and trust must extend beyond source code to include the entire toolchain. By exposing this hidden vulnerability, Thompson laid the groundwork for understanding and addressing *software supply chain attacks* [Cesarano et al. 2024; Ladisa et al. 2023; Ohm et al. 2020; Okafor et al. 2024], a concern that remains relevant today.

Authors' Contact Information: Guilherme de Oliveira Silva, gsilva@cadence.com, Cadence, Belo Horizonte, Brazil; Fernando Magno Quintão Pereira, fernando@dcc.ufmg.br, UFMG, Brazil, fernando@dcc.ufmg.br.

The Challenge of Building Trust. In Thompson's word, instead of trusting that a program is free of Trojan horses, "*Perhaps it is more important to trust the people who wrote the software*" [Thompson 1984]. Nevertheless, various techniques have been developed to enhance trust in software and mitigate the risks of Thompson-style backdoors. One prominent approach specifically addressing this issue is the Diverse Double-Compiling technique, proposed by David A. Wheeler in his PhD dissertation [Wheeler 2005, 2010]. As Section 5.1 further explains, this method compares binaries produced by different compilers to identify malicious behavior in a potentially compromised compiler.

Going beyond Wheeler's certification methodology, programming language techniques provide broader mechanisms to build trust in software. Translation validation [Necula 2000; Pnueli et al. 1998] verifies that the output of a compiler faithfully preserves the semantics of the input program by validating each individual compilation rather than the entire compiler. Proof-carrying code (PCC) [Necula 1997] allows a program consumer to ensure that code provided by an untrusted producer satisfies specific safety properties without needing to trust the producer. Ultimately, compilers can be fully verified. For instance, CompCert [Leroy 2009] is a compiler whose correctness is formally guaranteed via the Coq proof assistant. Together, these techniques strengthen the foundations of trust in the software ecosystem. And yet, the fundamental issue of "*who we trust*" still plagues them all.

The aforementioned techniques – certified compilation (as in CompCert), Proof-Carrying Code (PCC), Translation Validation, and Diverse Double-Compiling (DDC) – each have limitations in addressing the fundamental challenge raised by Thompson's "Trusting Trust" attack. These limitations arise primarily because demonstrating that a binary faithfully implements the semantics of arbitrary source code requires solving the problem of semantic equivalence, which is undecidable in general due to its relationship to the Halting Problem [Church 1936; Turing 1936].

Proof-Carrying Code: PCC is designed to demonstrate specific safety properties, such as the absence of out-of-bounds memory accesses (e.g., ensuring all array accesses arr[i] satisfy $0 \leq i <$ size). While effective at verifying certain well-defined properties, PCC does not address the presence of malicious behavior, such as Trojan horses, in the binary.

Translation Validation: The validator verifies the correctness of individual compiler transformations, such as register allocation [Rideau and Leroy 2010] or constant propagation [Pnueli et al. 1999], rather than the entire compiler. By focusing on specific transformations, translation validation avoids undecidability. However, it is inherently limited to the scope of the program and transformations being validated. If validation fails, it may indicate a limitation in the validation technique rather than an actual error in the compiler.

Verified Compilation: Unlike PCC, a verified compiler such as CompCert does not produce standalone certificates that users can independently verify. Instead, CompCert provides strong semantic correctness guarantees for the programs it compiles, but it does not address the problem of trusting the compiler binary itself. If the binary of the verified compiler is compromised, all guarantees it provides are invalid.

Diverse Double-Compiling: As Section 5.1 explains, DDC relies on the assumption that at least one trusted compiler exists. While this technique reduces the likelihood of an undetected backdoor, it cannot eliminate the possibility entirely. If both compilers used in the DDC process are compromised, then Thompson's backdoor may still persist.

Therefore, each of these techniques represents a step toward building trust in software, but none can completely eliminate the risks highlighted by Thompson's seminal work.

The Contribution of This Work: This paper proposes a technique to certify that a target (low-level) program was produced as the result of translating a source (high-level) program. Said technique

is based on the famous encoding Kurt Gödel used in the proof of his incompleteness theorems [Gödel 1931]. As explained in Section 3, the certificate of faithful compilation is embedded within the structure of the high-level program and in the structure of the low-level program, rather than existing as a separate artifact, as is the case with proof-carrying code. Figure 1 illustrates the proposed methodology.
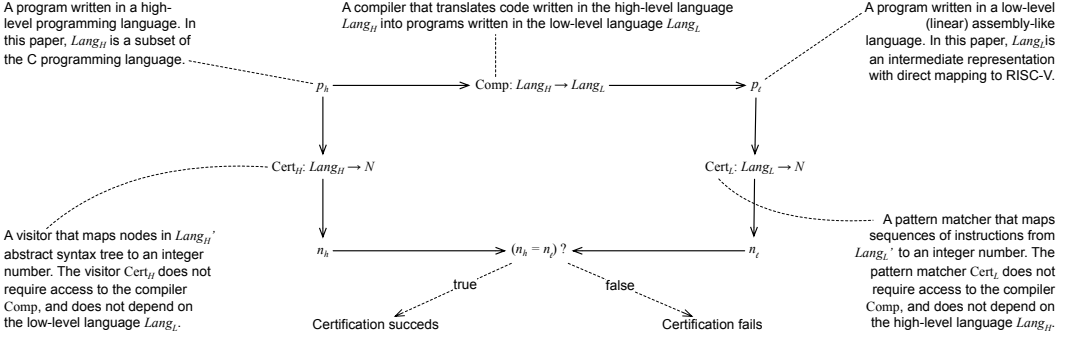

Fig. 1. Overview of the proposed certification methodology.

In Figure 1, p_h is a program written in a high-level source language $Lang_H$, and p_ℓ is the program written in a low-level target language $Lang_L$, produced by a compiler $Comp$, i.e., $p_\ell = Comp(p_h)$. Figure 1 defines two mapping algorithms, $H : Lang_H \mapsto \mathbb{N}$ and $L : Lang_L \mapsto \mathbb{N}$. If $Cert_H(p_h) = Cert_L(Comp(p_h))$, then $Comp$ has not modified the compilation contract, meaning that $Comp$ has generated only the instructions necessary to implement in p_ℓ the semantics of p_h. Example 1.1 will clarify these ideas.

Example 1.1. Figure 2 shows how we produce a certificate for a piece of code that is often considered the shortest valid C program. This example illustrates some basic principles behind our approach. First, a certificate is a product of powers of prime numbers. The base of such exponents, e.g., numbers such as 2, 3 and 5 in Figure 2 mark the position where each syntactic construct appears (first, second, third, etc) in the program's AST, or in the low-level intermediate representation (IR) that we adopt in this paper. The exponents are associated with particular programming constructs in the high-level language, or with particular sequences of instructions in the low-level language. For instance, a return statement in the high-level language is associated with the exponent 19. The code that corresponds to a return statement in the low-level language uses two opcodes: MOV and JR. The certificate of a program is the product of the certificate of all the constructs that make up that program. This sequence is also associated with exponent 19. In this example, this certificate is $2^2 \times 3^{19} \times 5^{113}$. In this paper we adopt as the low-level language a linear intermediate representation (IR) that has one-to-one correspondents with RISC-V instructions, as Figure 2 shows. Thus, by certifying the intermediate representation, we also certify its corresponding RISC-V image.

Example 1.1 illustrates how the proposed methodology works on a very simple program. This example misses two capacities of this approach: the ability to certify high-level constructs that are translated to non-contiguous sequences of instructions (like loops and branches); and the ability to encode def-use relations, as we would have once the high-level program manipulates variables. Section 3 will explain how we deal with such constructs.

Summary of Results. We have implemented the ideas proposed in this paper in a compilation framework called CHARON. This compiler is capable of certifying a subset of C that is expressive
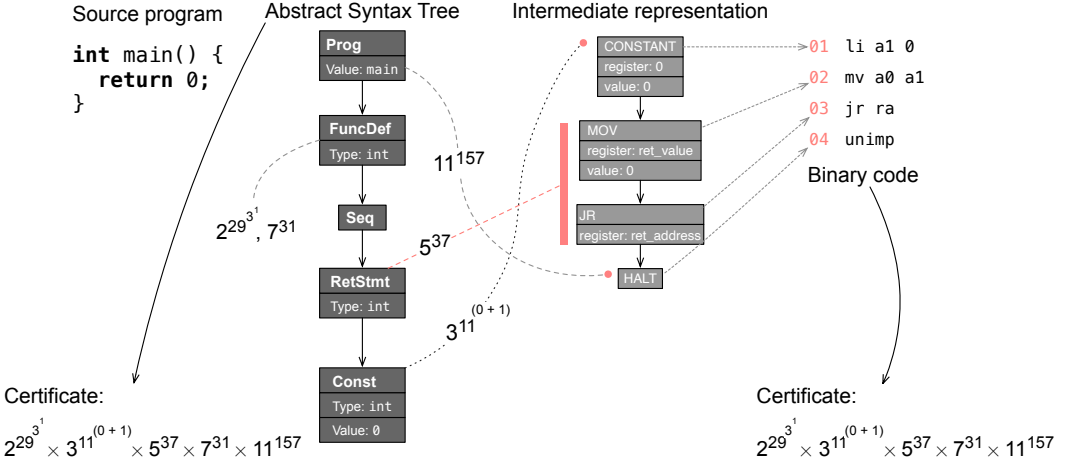
Fig. 2. How Charon generates certificates for a simple C program.

enough to support FaCT [Cauligi et al. 2019] programs. FaCT is a restricted version of C designed to facilitate constant-time programming. It provides users with three control-flow constructs: if-then-else, while, and return, along with a type system that allows for the specification of public and classified data. Charon supports all the control-flow constructs in FaCT, three data types (int16, int32, and float), and implicit type coercions. Henceforth, we call this version of FaCT (the $Lang_H$ in Figure 1), CharonLang. Similarly, we name CharonIR our target low-level intermediate representation (the $Lang_L$ in Figure 1).

Our certification methodology provides the guarantees seen in Figure 1, plus *invertibility*. The invertibility of the high-level certification process means that there exists a function $Canon : \mathbb{N} \mapsto Lang_H$, such that if $Cert_H(P_H) = n$, then $Canon(n) = P'_H$, where P'_H is the *Canonical Representation* of program P_H. As we explain in Section 3.4, P_H and P'_H are not expected to be syntactically equivalent, but they are expected to be semantically equivalent.

The absolutely value of a certificate, e.g., measured as the number of bits necessary to represent it, grows exponentially with the size of programs. This asymptotic behavior characterizes $Cert_H$ and $Cert_L$. In the former case, because a certificate contains a multiplicative factor for each construct in the program's AST. In the latter, because the certificate contains a multiplicative factor for each instruction in the program's low-level representation. Nevertheless, as we show in Section 4, the symbolic representation of a certificate, as a string that encodes a series of multiplications, is still linear on the size of the program, be it written in CharonLang or in CharonIR.

2 Trusting Trust

This section provides the background knowledge necessary to follow the ideas that will be developed in Section 3. To this end, Section 2.1 starts explaining how Thompson's hack works. Section 2.2, in turn, discusses the principles behind Gödel's Numbering System.

2.1 The Thompson Hack

Ken Thompson's famous "Trusting Trust" hack demonstrates a self-replicating backdoor embedded into a compiler – the "Trojan Compiler". Thompson's exploit starts with a clean compiler, which does not contain any backdoors. This compiler is trusted by the user and produces backdoor-free

binaries. The attack leverages two backdoors to propagate malicious behavior, even when the source code appears clean. The process can be described in four stages, as illustrated in Figure 3.

Fig. 3. A schematic view of Thompson's "Trusting Trust" attack.
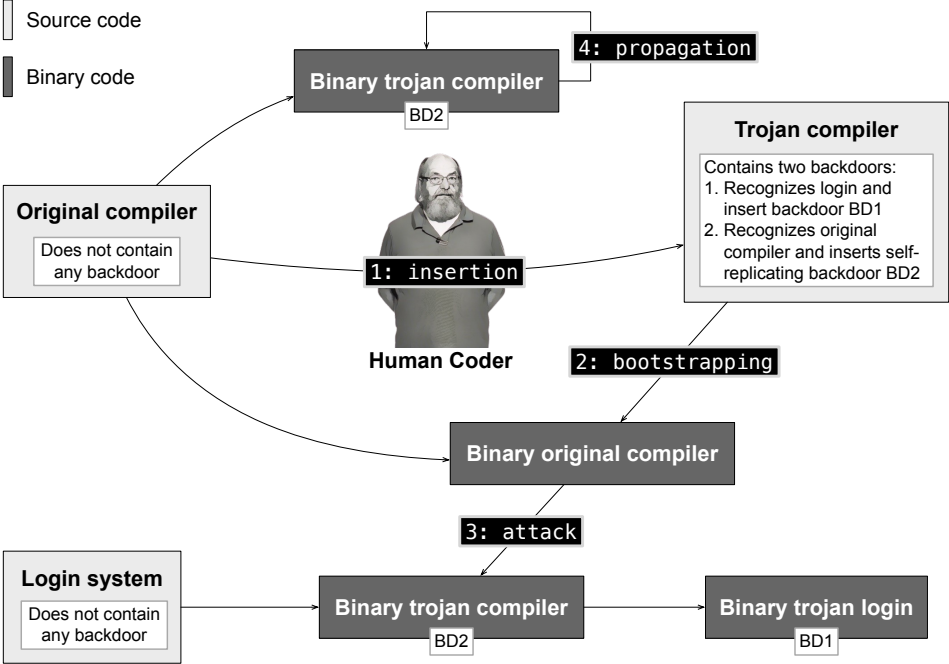
(1) **Insertion:** In Thompson's hack, a (malicious) user modifies the compiler source to produce a "trojan compiler", which contains two backdoors:
- **BD1:** This backdoor recognizes when the trojan compiler is compiling the login program and inserts a malicious backdoor (e.g., a hardcoded password) into the compiled binary.
- **BD2:** This self-replicating backdoor recognizes when the trojan compiler is compiling its own source code and reinserts both BD1 and BD2 into the new compiler binary.

(2) **Bootstrapping:** Once the trojan compiler's source is compiled with the original compiler, it produces a binary version of the trojan compiler. This binary now contains both backdoors, BD1 and BD2. At this point, the source code of the trojan compiler is no longer necessary, and can be removed. Therefore, the source code of the compiler appears clean, as BD2 exists only at the binary level.

(3) **Attack:** When the binary trojan compiler compiles the login system, it inserts BD1 into the resulting binary, creating a login login system.

(4) **Propagation:** When the binary trojan compiler compiles a clean version of its own source code, it reinserts BD1 and BD2 into the new binary compiler. This ensures the trojan behavior propagates indefinitely, even if the source code is reviewed and found to be clean.

The genius of this attack lies in its self-propagating nature: the malicious behavior becomes embedded at the binary level and is undetectable through inspection of the source code alone. Trusting the compiler binary is therefore essential, as the hack demonstrates how source code auditing alone cannot guarantee the absence of malicious behavior.

2.2 Gödel Numbering

We aim to certificate programs by assigning unique identifiers to each instruction that encodes the operation, its operands, and its relative position in the program simultaneously. These identifiers are then combined to produce a single number that uniquely represents a program in such manner that any changes made to the order of operations or operands will produce a different certificate. This is accomplished with a numbering system based on *Gödel Numbers*. This numbering system works according to the following sequence of steps, which Example 2.1 will illustrate:

(1) **Assign Unique Numbers to Symbols:** Create an *encoding table* where each symbol in the logical system (e.g., variables, operators, parentheses) is assigned a unique integer.
(2) **Map Each Position to a Prime Number:** Use the sequence of prime numbers $(2, 3, 5, 7, \ldots)$ to represent the positions of symbols in the expression. The n-th symbol in the expression corresponds to the n-th prime number.
(3) **Calculate the Contribution of Each Symbol:** For each symbol s_i at position i:
   - Take the i-th prime p_i as the *base*.
   - Use the integer assigned to s_i from the encoding table as the *exponent*.
   - Compute $p_i^{e_i}$, where e_i is the encoded value of s_i.
(4) **Multiply All Contributions:** Compute the Gödel number by multiplying the contributions of all symbols:

$$G = \prod_{i=1}^{n} p_i^{e_i}$$

where n is the total number of symbols in the expression.

Example 2.1. Figure 4 shows how we can encode the expression $a + b$ using a Gödel Numbering System. Let us assume that our example system assigns the following numbers to each potential symbol in our arithmetic system: $a \mapsto 4, + \mapsto 3, b \mapsto 7$. Similarly, let us assume that the order in which symbols appear in the logic expression are associated with the primes in ascending order, e.g.: $1^{st} \mapsto 2, 2^{nd} \mapsto 3, 3^{rd} \mapsto 5, \ldots$. Given these assumptions, we compute the contribution of each one of the three symbols in the expression as follows: $2^4, 3^3$ and 5^7. The final encoding is the product of all these parcels, e.g., $16 \times 27 \times 78, 125 = 33, 750, 000$.

Expression | Encoding | Application

a + b

Symbol	Numbering
a	4
+	3
b	7

Base: 2   3   5

Expn: a   +   b

Num: 2^a 3^+ 5^b $=$ $2^4 \times 3^3 \times 5^7$ $=$ 33,750,000

Fig. 4. An example of encoding based on Gödel Numbers.

Gödel's numbering relies on the Fundamental Theorem of Arithmetic [Gauss 1966], which states that every integer has a unique prime factorization. This ensures that each Gödel number maps uniquely to one expression and that the original expression can be reconstructed by factorizing the number. Thus, the Gödel Numbering System achieves two properties: *uniqueness* and *reversibility*. The numbering system proposed in this work also meets these two properties.

3 Program Certification

Our solution to the *trusting trust* problem is implemented by the CHARON compiler. It includes a toy language, CHARONLANG, that covers a large subset of the C language, and a compiler that

targets an intermediate representation (CHARONIR) inspired by the RISC-V architecture [Asanović and Patterson 2014].

CHARONLANG. The CHARON toy language supports all the control-flow constructs present in FACT, the Flexible and Constant Time cryptographic programming language [Cauligi et al. 2019], namely forward and backward branches (loops) plus non-recursive function calls. Additionally, this subset of the C language supports integers (16-bit short, 32-bit int) and 32-bit floating point (float) types, plus (implicit) type casts[1] and all the ANSI C unary and binary mathematical, logical, and bit-wise operations. CHARONLANG also supports static arrays and user-defined structures that combine built-in types. Similarly to FACT, we chose not to add support to pointer arithmetic to CHARONLANG. Figure 5 presents its grammar specification.

Procedure definitions

func. def.	::=		
	|	$\theta\, f\, (\theta\, x)\{S\}$	internal procedure

Statements

S	::=		
	|	$S; S$	sequence
	|	$\theta_p\, x$	variable declaration
	|	$f(e)$	procedure call
	|	$x = e$	assignment
	|	**if** $(e)\ \{S\}$	if conditional
	|	**if** $(e)\ \{S\}$ **else** $\{S'\}$	if...else conditional
	|	**while** $(e)\ \{S\}$	while loop
	|	**return** e	procedure return

Expressions

e	::=		
	|	n	numeric literal
	|	x	variable
	|	$\Theta\, e$	unary operation
	|	$e\, \Theta\, e$	binary operation
	|	$x[e]$	array access
	|	$x.y$	struct access

Types

θ	::=		
	|	**short** | **int** | **float**	built-in types
	|	$\theta\, x[n]$	arrays
	|	**struct** $\{ \theta\, x; \theta\, y; \dots \}$	user-defined structures

Fig. 5. The CHARON Toy Language grammar.

Example 3.1. Figure 6 (a) shows the CHARONLANG implementation of a program that computes the maximum common divisor of two integers. Part (b) of that figure shows the program's abstract syntax tree. The compiler (*Comp* in Fig. 1) visits each node of the AST to produce the low-level program representation seen in Figure 6 (c). Similarly, the high-level certification algorithm (*Cert$_H$* in Fig. 1) visits the AST to produce the program's certificate, as Section 3.2 will explain.

---

[1]The compiler emits adequate type cast instructions to compatibilize operands. Users can not explicitly make type coercion operations.
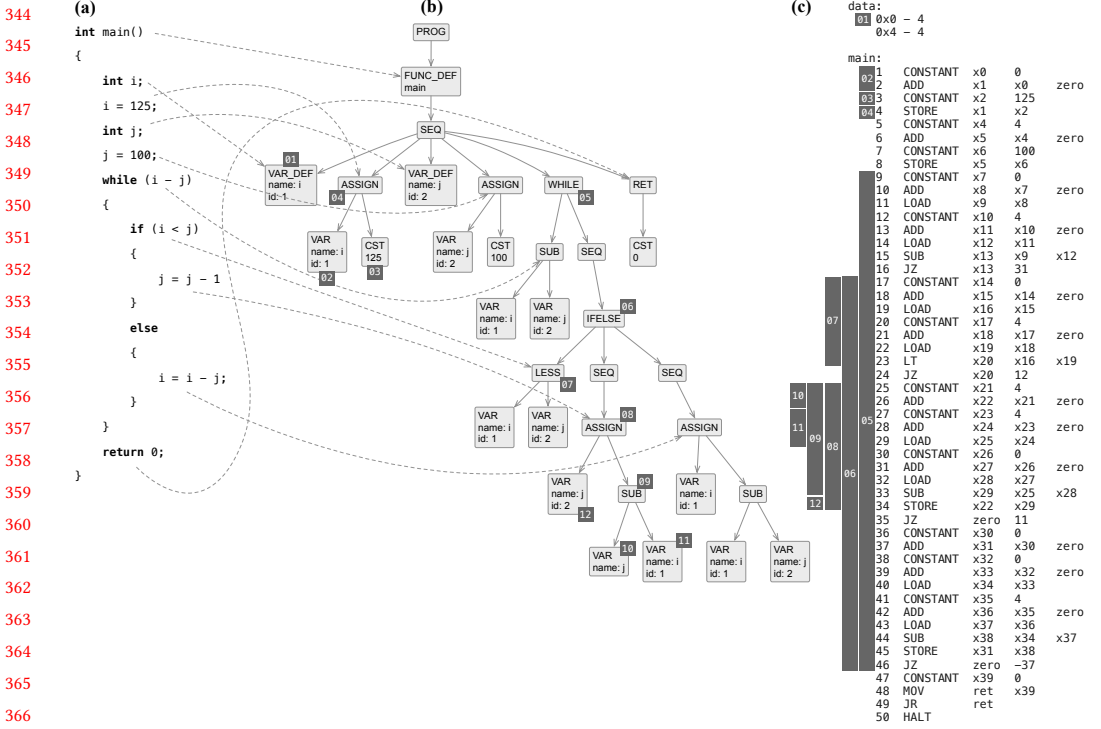
Fig. 6. (a) Maximum common divisor implemented in CHARONLANG (b) Abstract syntax tree. (c) Low-level representation of the algorithm written in CHARONIR.

Intermediate Representation. The CHARON Intermediate Representation, CHARONIR, uses the set of instructions presented in Figure 7. Notice that the IR features floating-point variations for all the operations, except the bit-wise ones. These variations have been omitted for brevity. Most instruction parameters are registers, except for imm, which takes a constant literal.

Example 3.2. Figure 6 (c) shows the three-address code representation of the implementation of maximum common divisor seen in Figure 6 (a). This example highlights that any value, variable or constant must be loaded into a temporary register before use. CHARONIR provides the CONSTANT instruction to load constants, because this explicit handling of literals simplifies the construction of their certificates, as Section 3.3 will explain. Retrieving the contents of a variable with LOAD requires the compiler to save its address in a temporary register first. This is achieved in two steps: first, it will emit a CONSTANT with its temporary address; then, it will compute the offset, if any, and store the final address in the register to load from. Following RISC-V, an object file is exported with minimal metadata; namely, a data section that describes the lengths of variables identified by their base addresses.

As Example 3.2 illustrates, CHARON's intermediate representation follows the RISC-V syntax. The primary difference from RISC-V is that CHARONIR has access to an infinite number of register names. Thus, CHARONIR registers should be understood as memory locations instead of typical registers. Despite this, the IR includes some predefined registers, adhering to the typical RISC-V/Linux Application Binary Interface. These predefined registers include one for storing the return address of functions, others for holding function arguments and return values, and a dedicated register that

Operations

ADD RD R1 R2	Addition	AND RD R1 R2	Logical and
BITAND RD R1 R2	Bit-wise and	BITOR RD R1 R2	Bit-wise or
DIV RD R1 R2	Division	EQ RD R1 R2	Equality
GT RD R1 R2	Greater than	LSHIFT RD R1 R2	Left bit shift
LT RD R1 R2	Less than	MOD RD R1 R2	Modulo
MULT RD R1 R2	Multiplication	NEQ RD R1 R2	Not equal
NOT RD R1	Logical negation	OR RD R1 R2	Logical or
RSHIFT RD R1 R2	Right bit shift	SUB RD R1 R2	Subtraction

Variables and constants

CONSTANT RD IMM$_{value}$	Load constant value
LOAD RD R1	Load variable value to register
STORE RD R1	Store value in address
MOV R1 R2	Move between registers

Type casts — **Control Flow**

FPTOSI RD R1	float → int	JAL LABEL	Jump-and-link to label
SIGNEXT RD R1	short → int	JR R1	Jump to register
SITOFP RD R1	int → float	JZ R1 IMM$_{size}$	Branch if zero
TRUNC RD R1	int → short	HALT	Terminate execution

Fig. 7. The CHARON intermediate representation. We let RD be destination and R1/R2 be source registers.

always contains zero for convenience. Variables in CHARONIR can be defined at multiple allocation sites, meaning the IR does not conform to the static single-assignment (SSA) format [Cytron et al. 1991], so pervasive in modern compiler design.

Translation Specification. Intermediate Representation code is generated by parsing CHARONLANG following the translation rules in Figure 8. Figure 8 uses a runtime environment T that maps variables to memory addresses. The n-th declared variable x will be mapped to its memory address, e.g., such as $T(x) \rightarrow address(n)$. Whenever a new variable or function is declared, the environment T is updated to a new T' that includes this definition. This mapping is necessary to distinguish variables that have the same name, when generating code (as seen in Figure 8), and when generating certificates.

3.1 The Numbering Schema

Given a CHARONLANG program P written with n symbols, its certificate $Cert(P)$ is the product of the first n primes, each raised to a particular exponent. Each symbol uses a different exponent, as we have already illustrated in Examples 1.1 and 2.1. Figure 9 shows the mapping of symbols to exponents.

Example 3.3. Figure 9 shows that an IF-THEN construct will be always associated with a triple of exponents $(41, 43, 47)$. These exponents indicate, respectively, the condition the if-then-else command will evaluate, the start of the conditional code block, and the end of the conditional code block. In regards to if-then-else statements, we have chosen to assign explicit markers to the beginning and to the end of these blocks so the certificate can be inverted back into the original program. Section 3.4 discusses inversion in more detail.

In contrast to the encoding of control-flow statements, the encoding of constants, variables, and functions vary depending on which symbols are being referenced. These cases are discussed below:

$$\frac{n' = n + \text{len}(\theta) \quad T_{n'} = T_n + (v, n)}{T_n \vdash \text{Comp}(\theta\ v) \xrightarrow{d} T_{n'}}$$

$$\frac{T \vdash \text{Comp}(v) \xrightarrow{a} I_a, \mathsf{R}_v \quad \text{fresh}(\mathsf{R})}{T \vdash \text{Comp}(v.x) \xrightarrow{a} I_a; \mathsf{ADD\ R\ R}_v\ T(v.x), \mathsf{R}}$$

$$\frac{i' = i + \text{len}(f) \quad T_{i'} = T_i + (f, i)}{T_i \vdash \text{Comp}(\theta_f\ f) \xrightarrow{d} T_{i'}}$$

$$\frac{\text{fresh}(\mathsf{R})}{T \vdash \text{Comp}(n) \xrightarrow{e} \mathsf{CONSTANT\ R}\ n, \mathsf{R}}$$

$$\frac{T \vdash \text{Comp}(S) \xrightarrow{s} I, T' \quad T' \vdash \text{Comp}(S') \xrightarrow{s} I', T''}{T \vdash \text{Comp}(S; S') \xrightarrow{s} I; I', T''}$$

$$\frac{T \vdash \text{Comp}(e) \xrightarrow{e} I_e, \mathsf{R}_e \quad \text{fresh}(\mathsf{R})}{T \vdash \text{Comp}(\Theta\ e) \xrightarrow{e} I_e; \mathsf{UnOp}_\Theta\ \mathsf{R}\ \mathsf{R}_e, \mathsf{R}}$$

$$\frac{T \vdash \text{Comp}(e_0) \xrightarrow{a} I_0, R_0 \quad T \vdash Comp(e_1) \xrightarrow{e} I_1, \mathsf{R}_1 \quad \text{fresh}(\mathsf{R})}{T \vdash \text{Comp}(e_0 = e_1) \xrightarrow{s} I_0; I_1; \mathsf{STORE\ R}_0\ \mathsf{R}_1, T}$$

$$\frac{T \vdash \text{Comp}(e_0) \xrightarrow{e} I_0, R_0 \quad T \vdash \text{Comp}(e_1) \xrightarrow{e} I_1, \mathsf{R}_1 \quad \text{fresh}(\mathsf{R})}{T \vdash \text{Comp}(e_0\ \Theta\ e_1) \xrightarrow{e} I_0; I_1; \mathsf{BinOp}_\Theta\ \mathsf{R\ R}_0\ \mathsf{R}_1, \mathsf{R}}$$

$$\frac{\text{fresh}(\mathsf{R}) \quad \text{fresh}(\mathsf{R}_v)}{T \vdash \text{Comp}(v) \xrightarrow{a} \mathsf{CONSTANT\ R}\ T(v); \mathsf{ADD\ R}_v\ \mathsf{R\ R}_{zero}, \mathsf{R}_v}$$

$$\frac{T \vdash \text{Comp}(e) \xrightarrow{e} I_e, \mathsf{R}_e \quad T \vdash \text{Comp}(S) \xrightarrow{s} I_t, T'}{T \vdash \text{Comp}(if(e)\ \{S\}) \xrightarrow{s} I_e; \mathsf{JZ\ R}_e\ len(I_t); I, T'}$$

$$\frac{T \vdash \text{Comp}(v) \xrightarrow{a} I_a, \mathsf{R}_v \quad T \vdash \text{Comp}(e) \xrightarrow{e} I_e, \mathsf{R}_e \quad \text{fresh}(\mathsf{R})}{T \vdash \text{Comp}(v[e]) \xrightarrow{a} I_a; I_e; \mathsf{ADD\ R\ R}_v\ \mathsf{R}_e, \mathsf{R}}$$

$$\frac{T \vdash \text{Comp}(if(e)\ \{S\}) \xrightarrow{s} I_t, T' \quad T' \vdash \text{Comp}(S') \xrightarrow{s} I_f, T''}{T \vdash \text{Comp}(if(e)\ \{S\}\ else\ \{S'\}) \xrightarrow{s} I_t; \mathsf{JZ\ zero}\ len(I_f); I_f, T''}$$

$$\frac{\text{fresh}(\mathsf{R}) \quad T \vdash \text{Comp}(e) \xrightarrow{a} I, \mathsf{R}_a}{T \vdash \text{Comp}(e) \xrightarrow{e} I; \mathsf{LOAD\ R\ R}_a, \mathsf{R}}$$

$$\frac{T \vdash \text{Comp}(e) \xrightarrow{e} I_e, \mathsf{R}_e}{T \vdash \text{Comp}(return\ e) \xrightarrow{e} I_e; \mathsf{MOV\ R}_{ret}\ \mathsf{R}_e; \mathsf{JR\ R}_{addr}, \mathsf{R}_e}$$

$$\frac{T \vdash \text{Comp}(e) \xrightarrow{e} I_e, \mathsf{R}_e \quad T \vdash \text{Comp}(S) \xrightarrow{s} I_t, T'}{T \vdash \text{Comp}(while(e)\ \{S\}) \xrightarrow{s} I_e; \mathsf{JZ\ R}_e\ len(I_t); I_t; \mathsf{JZ\ zero} - (len(I_t) + len(I_e) + 1), T'}$$

$$\frac{T_n \vdash \text{Comp}(\theta_p\ p) \xrightarrow{d} T_{n'} \quad T_{n'} \vdash \text{Comp}(S) \xrightarrow{s} I_b, T_{n''} \quad T_i \vdash \text{Comp}(\theta_f\ f) \xrightarrow{d} T_{i'} \quad \text{fresh}(\mathsf{R})}{T_{n,i} \vdash \text{Comp}(\theta_f\ f(\theta_p\ p)\{S\}] \xrightarrow{s} f: \mathsf{CONSTANT\ R}\ n; \mathsf{STORE\ R\ R}_{arg}; I_b, T_{i',n''}}$$

$$\frac{T \vdash Comp(e) \xrightarrow{e} I_e, \mathsf{R}_e \quad \text{fresh}(\mathsf{R}) \quad addr = num\_inst + len(I) + 2}{T \vdash \text{Comp}(f(e)) \xrightarrow{e} I_e; \mathsf{MOV\ R}_{arg}\ \mathsf{R}_e; \mathsf{CONSTANT\ R}_{addr}\ addr; \mathsf{JAL\ f}; \mathsf{MOV\ R\ R}_{ret}, \mathsf{R}}$$

Fig. 8. Specification of the translation rules.

Constant: A constant c is associated with the exponent $11^{c'}$, where c' is $c + 1$ if $c \geq 0$, or c otherwise. Therefore, each constant is represented by a unique value. Adding one is necessary to avoid the identity case of the exponentiation.

Variable definition: Variables definitions encoded with the base number 13 to the power of *var. def. exp.*, following the formula *var. def. exp.* = *type symbol*$_1^{type\ symbol_2^{\cdots}}$, where *type symbol*$_n$ is the symbol associated with the type of the n-th element of the variable. Scalars will have a single *type symbol*. Function parameters are encoded similarly, but use 23 as the base number.

Variable usage: Variable usage cases employ the base number 17, to the power of *var. usage exp.*, which considers the variable prime vp and the memory offset of this element from the variable's base address. This exponent comes from the formula *var. usage exp.* = $vp^{mem.\ offset}$. If the memory offset is static (i.e., indexing an array with a constant, or accessing an element from a *struct*), then *mem. offset* = 2^{os+1}, where os is the offset in bytes. Scalar variables are

Types			Variables and constants	
SHORT	2		CONSTANT	$11^{c'}$
INT	3		VAR. DEF.	$13^{var.\ def.\ exp.}$
FLOAT	5		VAR. USAGE	$17^{var.\ usage.\ exp.}$
__UNKNOWN_TYPE__	7			

Functions				
FUNC. PARAM.	$19^{var.\ def.\ exp.}$		FUNC. (START)	$31^{func.\ def.\ exp.}$
FUNC. ARG.	23		FUNC. (END)	37
FUNC. CALL	29^{fp}		FUNC. RETURN	41

Control Flow				
CONDITIONAL	43		ELSE (END)	59
IF (START)	47		WHILE (START)	61
IF (END)	53		WHILE (END)	67

Operations				
ASSIGN (=)	71		EQUAL (==)	109
NEGATION (!)	73		NOT EQUAL (! =)	113
ADDITION (+)	79		LOGICAL AND (&&)	127
SUBTRACTION (−)	83		LOGICAL OR (||)	131
MULTIPLICATION (∗)	89		LEFT SHIFT (<<)	137
DIVISION (/)	97		RIGHT SHIFT (>>)	139
MODULO (%)	101		BIT-WISE AND (&)	149
LESS THAN (<)	103		BIT-WISE OR (|)	151
GREATER THAN (>)	107			

Miscellaneous	
HALT	157

Fig. 9. Mapping of programming constructs to the exponents used in the numbering schema.

covered by this case and have $os = 0$. If the memory offset is dynamically determined (as in indexing an array with a variable), then *mem. offset* $= 3^{vp_{id}}$, where vp_{id} is the variable prime of the index.

Function definition: Function definitions are associated with a pair of symbols: FUNC. (START) and FUNC. (END). These symbols explicitly delimit the beginning and the end of the function scope. While the latter uses 37 as the base number, the former is represented by $31^{func.\ def.\ exp.}$, following the formula *func. def. exp.* $=$ *type symbol*$(\theta)^{(num.\ params.(f)+1)}$ where *type symbol*(θ) is the symbol associated with the function type, and *num. params.* is the number of parameters the function takes. *num. params.* is incremented by 1 to avoid the identity case of exponentiation for the case when a function takes no parameters.

Function call: Function calls are encoded with a base number, 29, to the power of the function prime fp_m that corresponds to the called function.

3.1.1 *Identification of variables and functions.* The Numbering Schema uses unique prime numbers as a means to identify variables and functions. These numbers are determined by the variable's or function's declaration site, in a way that resembles De Bruijn's indexation [de Bruijn 1994].

We collect all the variables definitions − including function parameters −, in the order they are defined, and map the n-th active variable in this set to the n-th prime number vp_n. A variable is active if it has at least one use case, and we treat function parameters as always active. The prime of a variable is emitted as soon as an use case of it is found.

Function primes fp_m are produced by mapping the m-th function label to the m-th prime number fp_m. These numbers are emitted as soon as the function definition is parsed by the compilation algorithm $Comp$. Unlike variable primes, that only consider active variables, function primes are emitted regardless of whether the function is ever called.

Example 3.4. Consider the set of variables definitions $\{var_1, var_2, var_3\}$. Assume that var_1 and var_3 both have use cases, but var_2 does not. In this case, var_1 is mapped to the first variable prime $vp_1 = 2$, while var_3 is mapped to the second variable prime $vp_2 = 3$.

Variable primes vp and function primes fp are managed by a certification environment C. The high-level certification algorithm uses the C_H environment, which will be discussed in Section 3.2. C_H maps variables' aliases to variable primes, and function names to function primes. The low-level certification algorithm, on the other hand, is described in Section 3.3 and uses C_L. C_L maps variables' base addresses to variable primes, and the function labels to function primes. While these environments are fully independent, Figure 10 presents how they produce variable and function primes from the compilation process.

$$\frac{vp' = \text{next}(vp) \text{ if } v \notin C_H \quad C_H' = C_H \cup \{v \mapsto vp'\} \quad C_L' = C_L \cup \{T(v) \mapsto vp'\} \quad fresh(\mathsf{R}) \quad fresh(\mathsf{R}_v)}{(T, C_H, C_L, vp) \vdash \text{Comp}(v) \xrightarrow{a} \text{CONSTANT } \mathsf{R} \ T(v); \text{ADD } \mathsf{R}_v \ \mathsf{R} \ \mathsf{R}_{zero}, \mathsf{R}_v, \ C_H', C_L', vp'}$$

$$\frac{(T, C_H, C_L, vp) \vdash \text{Comp}(v) \xrightarrow{a} I_a, \mathsf{R}_v, C_H', C_L', vp' \quad T \vdash \text{Comp}(e) \xrightarrow{e} I_e, \mathsf{R}_e \quad fresh(\mathsf{R})}{(T, C_H, C_L, vp) \vdash \text{Comp}(v[e]) \xrightarrow{a} I_a; I_e; \text{ADD } \mathsf{R} \ \mathsf{R}_v \ \mathsf{R}_e, \mathsf{R}, C_H', C_L', vp'}$$

$$\frac{(T, C_H, C_L, vp) \vdash \text{Comp}(v) \xrightarrow{a} I_a, \mathsf{R}_v, C_H', C_L', vp' \quad fresh(\mathsf{R})}{(T, C_H, C_L, vp) \vdash \text{Comp}(v.x) \xrightarrow{a} I_a; \text{ADD } \mathsf{R} \ \mathsf{R}_v \ T(v.x), \mathsf{R}, C_H', C_L', vp'}$$

$$\frac{fp' = \text{next}(fp) \text{ if } f \notin C_H \quad C_H' = C_H \cup \{f \mapsto fp\} \quad C_L' = C_L \cup \{f \mapsto fp\}}{(T_i, C_H, C_L, fp) \vdash \text{Comp}(\theta_f \ f) \xrightarrow{d} \mathsf{f}, T_i, C_H', C_L', fp'}$$

Fig. 10. Emission of variable and function primes vp_n and fp_m.

Example 3.5. Figure 11 shows how the Numbering Schema handles functions and variables. As func_1 is the first function declared in this program, it will be identified by the first function prime, $fp_1 = 2$. The next declared functions (func_2 and main) are identified by $fp_2 = 3$ and $fp_3 = 5$, respectively. A similar logic applies to variables definitions, but, in this case, their types are also taken into consideration. The first variable to be declared is int param_1, from func_1. It has *type symbol* = 3 and will be identified by the first variable prime $vp_1 = 2$. The next variables are float param_2 ($vp_2 = 3$, *type symbol* = 5), and int some_var[3] ($vp_3 = 5$, *type symbol* = 7^{7^3}), and int var_2 ($vp_5 = 11$, *type symbol* = 3). some_var is an array. Only its third position is used in the program, and it is not indexed via variable indices; hence, only the third position consumes a prime in the certification process. The last line of the example shows an expression that takes a variable and the result of a function call as operands. To encode it, the Numbering Algorithm will use the addition base symbol (79), and will produce symbols to represent the variable usage and the function call. The variable usage symbol will be the base 17 to the power of $vp^{mem.offset}$, where *mem.offset* = 2^{0+1} = 2 and $vp_5 = 11$. Rearranging, the symbol will be $17^{11^2} = 17^{121}$. The function call symbol, in turn, will be $29^{fp_m} = 29^3$. The positional primes of each component have been omitted for brevity.
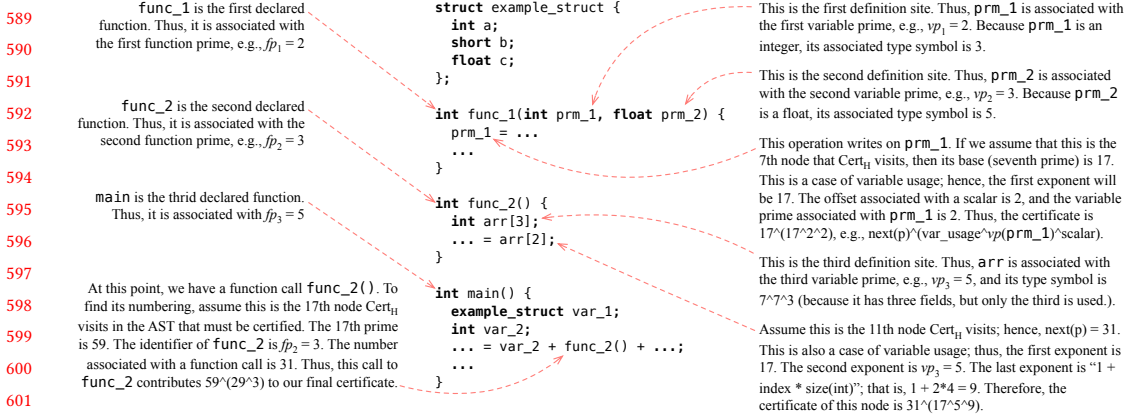
Fig. 11. Example illustrating the certification of variables and functions.

By associating a different prime to each variable, the certification schema is able to distinguish programs that access the same variables, but in different order. We notice that this schema does not take commutativity into consideration. Thus, $x + y$ and $y + x$ will produce different certificates. This property applies to both, the high-level certifier and the low-level certifier, for, according to Figure 11, both associate user-defined symbols with primes via the environments C_H and C_L.

3.2 Source Code Certification

The certification of programs written in the high-level language (CHARONLANG) happens in two phases. First, an environment C_H is created following the rules in Figure 10. Then, the certificates are produced by a function $Cert_H$, following the rules in Figure 12. Notice that these rules are parameterized by the environment C_H, and that we do not pass C_H as a parameter to $Cert_H$, because it never changes; rather, we treat C_H as an immutable global table.

In Figure 12, p represents the current positional prime, and the other values follow the Numbering Schema seen in Section 3.1. The certifier $Cert_H$ traverses the program's abstract syntax tree, assigning numbers to nodes in post-order. This is motivated by the fact that the compiler will generate code for the operands before emitting the instructions of the operations themselves. By certifying expressions in post-order, we conciliate the sequence of numbers produced by $Cert_H$ with the sequence produced by $Cert_L$.

After traversing the AST and computing the encoding numbers for all of its nodes, $Cert_H$ will move all the variable and parameter definition symbols to the beginning of the certificate in the order they appear in the original program, and recompute all of the positional primes. The relative position of the remaining numbers is also preserved. This is a key design choice that enables the inversion of certificates[2]. Inversion is discussed in Section 3.4.

Example 3.6. Figure 13 shows the certificate produced for a WHILE statement. This example assumes that x has $vp_1 = 2$, y has $vp_2 = 3$, and func has $fp_1 = 2$. Following the rules in Figure 12, $Cert_H$ first certificates the conditional in the loop, which is the binary operation $x < 10$. In this case, $Cert_H$ will first emit the COND symbol, and then compute the certificate of the left-hand side, the reading of variable x; the right-hand side, the constant 10; and finally the binary LESS operation itself. After that, $Cert_H$ emits the number associated with the start of a WHILE construct, which

---

[2]Adding symbols such as COND, IF (START), and IF (END) to explicitly mark the boundaries of each part of conditionals when handling them follows the same motivation.

Procedure definitions

$Cert_H(\theta\ f(\theta_p\ x, ...)\{S\})$ ::= $p^{31^{func.\ def.\ exp.}} \times Cert_H(\theta_p x) \times Cert_H(...) \times Cert_H(S) \times next(p)^{37}$;

     $func.\ def.\ exp. = type\ symbol(\theta)^{(num.\ params.(f)+1)}$

Statements

$Cert_H(S; S')$ ::= $Cert_H(S) \times Cert_H(S')$

$Cert_H(\theta\ x)$ ::= $p^{13^{type\ symbol(\theta)}}$

$Cert_H(\theta_p\ x)$ ::= $p^{19^{type\ symbol(\theta_p)}}$

$Cert_H(f(e))$ ::= $Cert_H(e) \times next(p)^{23} \times next(p)^{29^{C_H(f)}}$

$Cert_H(x = e)$ ::= $Cert_H(x) \times Cert_H(e) \times next(p)^{71}$

Control flow

$Cert_H(\textbf{if}\ (e)\ \{S\})$ ::= $p^{43} \times Cert_H(e) \times next(p)^{47} \times Cert_H(S) \times next(p)^{53}$

$Cert_H(\textbf{if}\ (e)\ \{S\}\ \textbf{else}\ \{S'\})$ ::= $Cert_H(\textbf{if}\ (e)\ \{S\}) \times Cert_H(S') \times next(p)^{59}$

$Cert_H(\textbf{while}(e)\ \{S\})$ ::= $p^{43} \times Cert_H(e) \times next(p)^{61} \times Cert_H(S) \times next(p)^{67}$

$Cert_H(\textbf{return}\ e)$ ::= $Cert_H(e) \times next(p)^{41}$

Expressions

$Cert_H(n)$ ::= $p^{11^{(n+1)\ if\ n\ \geq\ 0\ else\ n}}$

$Cert_H(x)$ ::= $p^{17^{var.\ usage\ exp.}}$; $var.\ usage\ exp. = C_H(x)^2$

$Cert_H(x[n])$ ::= $p^{17^{var.\ usage\ exp.}}$; $var.\ usage\ exp. = C_H(x)^{2^{1+n \times size(type(x))}}$

$Cert_H(x.y)$ ::= $p^{17^{var.\ usage\ exp.}}$; $var.\ usage\ exp. = C_H(x)^{2^{1+offset(y)}}$

$Cert_H(x[y])$ ::= $p^{17^{var.\ usage\ exp.}}$; $var.\ usage\ exp. = C_H(x)^{3^{C_H(y)}}$

$Cert_H(\Theta\ e)$ ::= $Cert_H(e) \times next(p)^{symbol(\Theta)}$

$Cert_H(e_1\ \Theta\ e_2)$ ::= $Cert_H(e_1) \times Cert_H(e_2) \times next(p)^{symbol(\Theta)}$

Fig. 12. Rules for the certification of programs in the high-level language CharonLang.

ends up as 11^{61}, i.e., the fifth prime raised to 61. In this case, 61 is the exponent associated with the beginning of a While block. Next, $Cert_H$ proceeds to build a certificate to the body of the while operation, similarly to what we saw in Example 3.5. Finally, it computes the number that represents the end of the conditional code, 59^{67}. The exponent 67 is associated with the end of a While block.

3.3 Machine Code Certification

The Machine Code Certification algorithm $Cert_L$ produces a certificate for the compiled program, which exists in the low-level CharonIR. The compiled program consists of a list of instructions for the Virtual Machine to execute, and a set of metadata — the data section, as previously introduced. As in the high-level case, the certification of low-level code happens in two phases. The first creates the environment C_L (Figure 10); whereas the second builds the certificate itself (Figure 14).

As already mentioned, Figure 10 shows the first phase of the certification process. The rules in Figure 10 produce a map between the base memory address of each variable and a unique variable prime vp, and add it to the runtime environment C_L. This environment also contains mappings between each function label to an unique function prime fp.

The second phase of the certification process, described in Figure 14, performs five actions:

(1) **Map Dependencies Between Instructions And Temporary Registers:** Produce a mapping between instructions and the temporary registers they depend upon. For instance, it will map the instruction of a binary operation to the temporary registers that contains its operands. This process is recursive. As such, if one of the operands of a binary operation Op_1
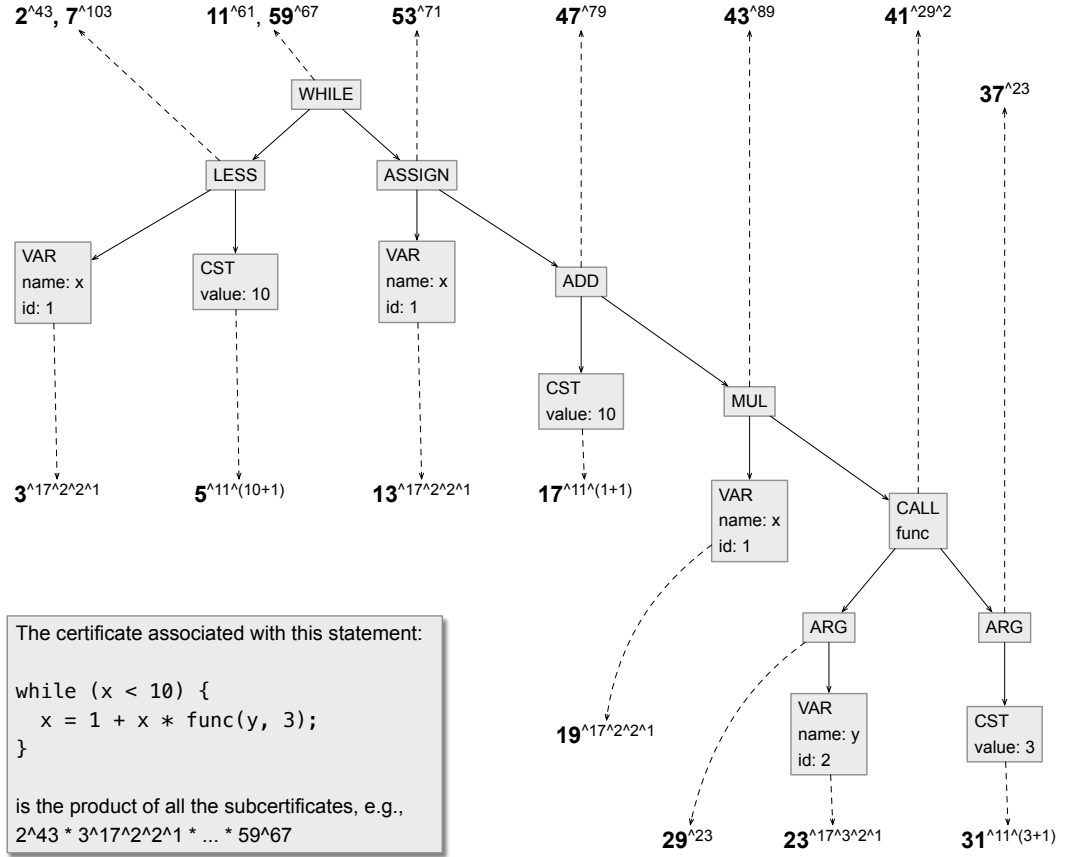
Fig. 13. Certification of a WHILE statement, i.e., $Cert_H(\textbf{while}(e) \ \{S\}) ::= p^{43} \times Cert_H(e) \times next(p)^{61} \times Cert_H(S) \times next(p)^{67}$.

is the result of another binary operation Op_2, Op_1 dependency mapping will also include the registers Op_2 depends upon.

(2) **Infer Variables Types:** Produce a mapping between variables addresses and their inferred types. Types are inferred based on the sequence of instructions used to read or write a variable. Reading from and writing to float variables will use the LOADF/STOREF instructions, respectively. The integer types, int and short, will use LOAD and STORE — the latter will also be truncated, with the TRUNC instruction, immediately after being read or before being written.

(3) **Analyze Functions:** Produce a mapping between functions labels and their inferred return types and number of parameters. Function type inference is computed from inferring the type of the value they return, following the same logic applied to infer variables types. The number of parameters it takes, on the other hand, is obtained from the counting the occurrences of the parameter passing pattern[3] within the function scope. This pass will also set up the insertion points of the FUNC. (START) and FUNC. (END) symbols in the program certificate, based on the scope each function label delimits.

---

[3]CONSTANT R_c $var_{address}$; STORE R_c R_{arg}

(4) **Analyze Conditional Jumps:** Search for conditional jumps (i.e., the JZ instruction) and determine the first instruction that is used in its condition. This pass is based on the previously computed dependency mapping, and it sets up the insertion point of the COND symbol in the program certificate.

(5) **Certificate Instructions:** Visit each instruction of the machine code to compute the adequate number following the rules if Figure 14. This figure presents the patterns the low-level certifier $Cert_H$ will attempt to match. The certifier will then proceed to the first instruction after the instructions in the pattern it just matched until it reaches the end of the program. This follows the design of certificating constructs, rather than individual instructions.

Statements

$$Cert_L(S; S') ::= Cert_L(S) \times Cert_L(S')$$

$$Cert_L(\texttt{data}[var_{address}]) ::= p^{13^{type\ symbol(var_{address})}}$$

$$Cert_L(\texttt{label}; S) ::= p^{31^{func.\ def.\ exp.}} \times Cert_L(S) \times next(p)^{37};$$
$$func.\ def.\ exp. = type\ symbol(type(\texttt{label}))^{(num.\ params.(\texttt{label})+1)}$$

$$Cert_L(\texttt{CONSTANT R}_c\ var_{address}; \texttt{STORE R}_c\ \texttt{R}_{arg}) ::= p^{17^{type\ symbol(var_{address})}}$$

$$Cert_L(S; \texttt{MOV R}_{arg}\ \texttt{R}) ::= Cert_L(S) \times next(p)^{23}$$

$$Cert_L(\texttt{CONSTANT R}_{addr}\ addr; \texttt{JAL label}; \texttt{MOV R R}_{ret}) ::= p^{29^{C_L(\texttt{label})}}$$

$$Cert_L(S; S'; \texttt{STORE R}_S\ \texttt{R}_{S'}) ::= Cert_L(S) \times Cert_L(S') \times next(p)^{71}$$

Control flow

$$Cert_L(S; \texttt{JZ R}_S\ len(S'); S') ::= p^{43} \times Cert_L(S) \times next(p)^{47} \times Cert_L(S') \times next(p)^{53}$$

$$Cert_L(S; \texttt{JZ R}_S\ len(S'); S'; \texttt{JZ zero}\ len(S''); S'') ::= p^{43} \times Cert_L(S) \times next(p)^{47} \times Cert_L(S') \times$$
$$next(p)^{53} \times Cert_L(S'') \times next(p)^{59}$$

$$Cert_L(S; \texttt{JZ R}_S\ len(S'); S';$$
$$\texttt{JZ zero} - (len(S) + len(S') + 1)) ::= p^{43} \times Cert_L(S) \times next(p)^{61} \times Cert_L(S') \times next(p)^{67}$$

$$Cert_L(S; \texttt{MOV R}_{ret}\ \texttt{R}_S; \texttt{JR R}_{addr}) ::= Cert_L(S) \times next(p)^{41}$$

Expressions

$$Cert_L(\texttt{CONSTANT R}_c\ \texttt{c}) ::= p^{11^c}$$

$$Cert_L(\texttt{CONSTANT R}_c\ \texttt{x}; \texttt{ADD R}_x\ \texttt{R}_c\ \texttt{zero}) ::= p^{17^{var.\ usage\ exp.}}; var.\ usage\ exp. = C_L(x)^2$$

$$Cert_L(\texttt{CONSTANT R}_c\ \texttt{x}; \texttt{ADD R}_x\ \texttt{R}_c\ \texttt{zero};$$
$$\texttt{CONSTANT R}_o\ offset; \texttt{ADD R}_{x+offset}\ \texttt{R}_x\ \texttt{R}_o) ::= p^{17^{var.\ usage\ exp.}}; var.\ usage\ exp. = C_L(x)^{2+offset}$$

$$Cert_L(\texttt{CONSTANT R}_c\ \texttt{x}; \texttt{ADD R}_x\ \texttt{R}_c\ \texttt{zero};$$
$$\texttt{CONSTANT R}_{c'}\ \texttt{y}; \texttt{ADD R}_y\ \texttt{R}_{c'}\ \texttt{zero};$$
$$\texttt{CONSTANT R}_{c''}\ \texttt{size(x)}; \texttt{MUL R}_{offset}\ \texttt{R}_y\ \texttt{R}_{c''};$$
$$\texttt{ADD R}_{x+offset}\ \texttt{R}_x\ \texttt{R}_{offset}) ::= p^{17^{var.\ usage\ exp.}}; var.\ usage\ exp. = C_L(x)^{3^{C_L(y)}}$$

$$Cert_L(S; \texttt{UnOp R}_{op}\ \texttt{R}_S) ::= Cert_L(S) \times next(p)^{symbol(\texttt{UnOp})}$$

$$Cert_L(S; S'; \texttt{BinOp R}_{op}\ \texttt{R}_S\ \texttt{R}_{S'}) ::= Cert_L(S) \times Cert_L(S') \times next(p)^{symbol(\texttt{BinOp})}$$

Fig. 14. Machine Code Certification rules.

Example 3.7. Figure 15 (a) displays the a simple program implemented in CHARONLANG that contains a WHILE loop. Part (b) of that figure presents its low-level representation in CHARONIR. The figure groups sequences of instructions into blocks to highlight the patterns that the certifier has matched to produce the final certificate. These blocks are mapped to part (d), which presents the certificate that encodes this program. Part (c) presents the environment C_L after it is populated by the rules in Figure 10, and the resulting maps of passes (1) through (3).
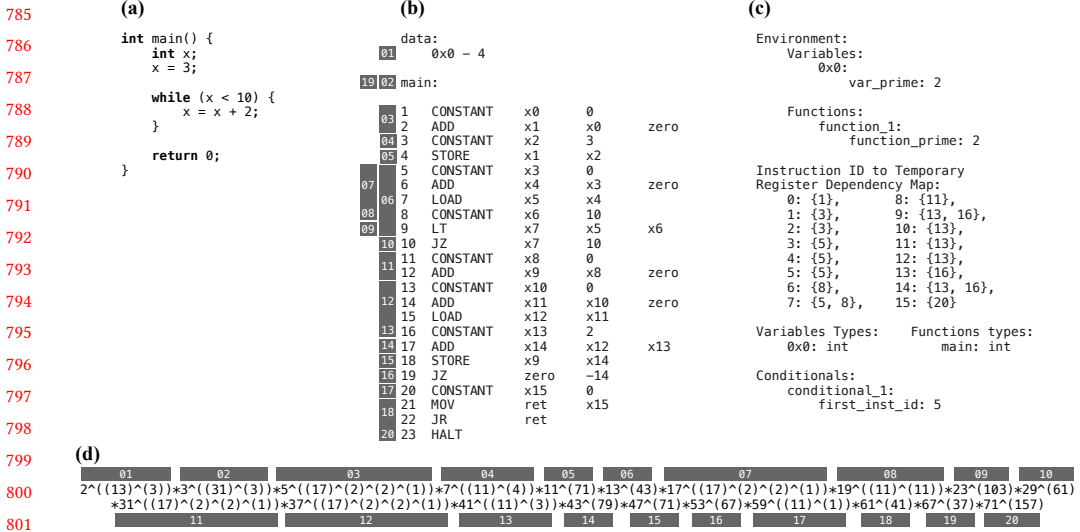
(a)
```
int main() {
    int x;
    x = 3;

    while (x < 10) {
        x = x + 2;
    }

    return 0;
}
```

(b)
```
data:
01    0x0 - 4
19 02 main:
03   1  CONSTANT  x0    0
04   2  ADD       x1    x0    zero
05   3  CONSTANT  x2    3
     4  STORE     x1    x2
     5  CONSTANT  x3    0
07   6  ADD       x4    x3    zero
06   7  LOAD      x5    x4
08   8  CONSTANT  x6    10
09   9  LT        x7    x5    x6
10  10  JZ        x7    10
11  11  CONSTANT  x8    0
    12  ADD       x9    x8    zero
    13  CONSTANT  x10   0
12  14  ADD       x11   x10   zero
    15  LOAD      x12   x11
13  16  CONSTANT  x13   2
14  17  ADD       x14   x12   x13
15  18  STORE     x9    x14
16  19  JZ        zero  -14
17  20  CONSTANT  x15   0
    21  MOV       ret   x15
18  22  JR        ret
20  23  HALT
```

(c)
```
Environment:
  Variables:
    0x0:
      var_prime: 2

  Functions:
    function_1:
      function_prime: 2

Instruction ID to Temporary
Register Dependency Map:
  0: {1},       8: {11},
  1: {3},       9: {13, 16},
  2: {3},      10: {13},
  3: {5},      11: {13},
  4: {5},      12: {13},
  5: {5},      13: {16},
  6: {8},      14: {13, 16},
  7: {5, 8},   15: {20}

Variables Types:    Functions types:
  0x0: int            main: int

Conditionals:
  conditional_1:
    first_inst_id: 5
```

(d)

01 02 03 04 05 06 07 08 09 10

$2^{((13)^{(3)})}*3^{((31)^{(3)})}*5^{((17)^{(2)}{}^{(2)}{}^{(1)})}*7^{((11)^{(4)})}*11^{(71)}*13^{(43)}*17^{((17)^{(2)}{}^{(2)}{}^{(1)})}*19^{((11)^{(11)})}*23^{(103)}*29^{(61)}$
$*31^{((17)^{(2)}{}^{(2)}{}^{(1)})}*37^{((17)^{(2)}{}^{(2)}{}^{(1)})}*41^{((11)^{(3)})}*43^{(79)}*47^{(71)}*53^{(67)}*59^{((11)^{(1)})}*61^{(41)}*67^{(37)}*71^{(157)}$

11 12 13 14 15 16 17 18 19 20

Fig. 15. Certification of a simple program with a WHILE loop.

3.4 The Canonical Format

Certificates are invertible. Thus, if $n = Cert_H(P) = Cert_L(Comp(P))$, then it is possible to reconstruct a program P_c from n. These two programs, P and P_c are semantically equivalent[4]; however, they are not syntactically equivalent. The program P_c is called the *canonical* representation of P. As such, the canonical form P_c has the same certificate as the program P it was reconstructed from. Notice that different programs might have the same canonical representation. If two programs have the same canonical representation, then they implement the same semantics. However, these programs might differ in how data is allocated. The following differences are possible:

Variables: All the variables are global, except for function parameters. The canonical form preserves the order in which variables are declared in the original program.

Structures: The canonical representation only considers data structures elements that are used in the program. However, it preserves the number of elements and their order in the original program. Thus, unused elements in the original program are still defined in the canonical representation, but they are typed as __unknown_type__. Arrays that are only indexed by constants in the original program are represented as structs in the canonical format. Arrays that are indexed with a variable, on the other hand, are still declared as arrays in the canonical format.

Example 3.8. Figure 16 presents two high-level programs, $P1$ and $P2$, such that $Cert_H(P1) = Cert_H(P2)$. While these programs are not syntactically identical, they possess the same semantics: they set the values of the first and third elements of a data structure — both integers — using constant offsets and return their sum. These programs differ only on how variables are stored. The second program stores variables as fields of a struct. The first does the same, albeit using an array. The canonical representation uses a struct to represent both situations. However, one of the fields

---

[4]In this paper, we do not show semantic equivalence between high- and low-level constructs: we only show proof that the compilation of one will yield the other. In our work, the only difference between programs that lead to the same canonical form is the declaration and storage of variables. Thus, it is simple to conclude that they would implement the same semantics.
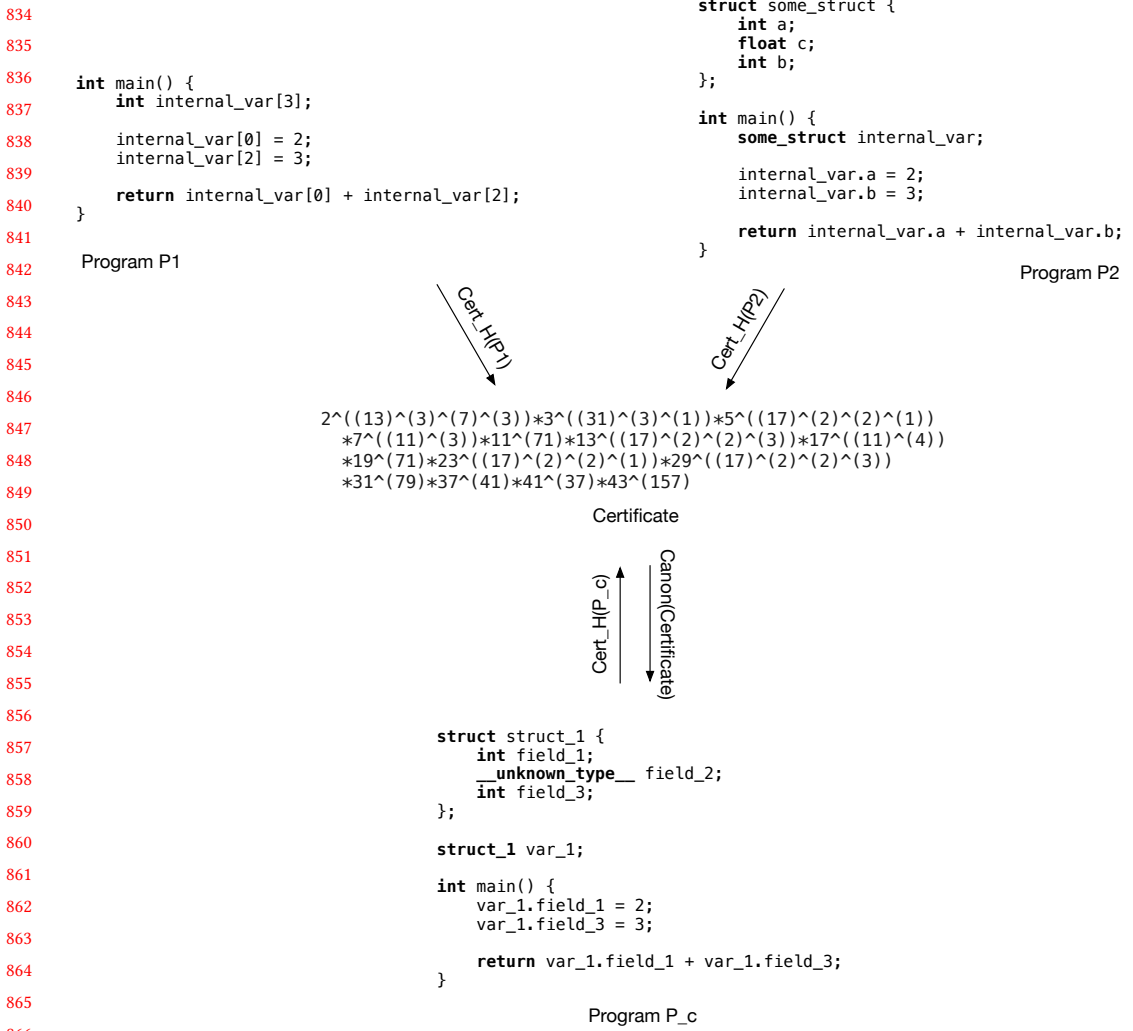
```c
struct some_struct {
    int a;
    float c;
    int b;
};

int main() {
    some_struct internal_var;

    internal_var.a = 2;
    internal_var.b = 3;

    return internal_var.a + internal_var.b;
}
```
Program P2

```c
int main() {
    int internal_var[3];

    internal_var[0] = 2;
    internal_var[2] = 3;

    return internal_var[0] + internal_var[2];
}
```
Program P1

Cert_H(P1)   Cert_H(P2)

2^((13)^(3)^(7)^(3))*3^((31)^(3)^(1))*5^((17)^(2)^(2)^(1))
*7^((11)^(3))*11^(71)*13^((17)^(2)^(2)^(3))*17^((11)^(4))
*19^(71)*23^((17)^(2)^(2)^(1))*29^((17)^(2)^(2)^(3))
*31^(79)*37^(41)*41^(37)*43^(157)

Certificate

Cert_H(P_c)   Canon(Certificate)

```c
struct struct_1 {
    int field_1;
    __unknown_type__ field_2;
    int field_3;
};

struct_1 var_1;

int main() {
    var_1.field_1 = 2;
    var_1.field_3 = 3;

    return var_1.field_1 + var_1.field_3;
}
```

Program P_c

Fig. 16. Inversion of the certificate of two semantically equivalent high-level programs.

of this struct is left undefined. Notice that this field still occupies 4 bytes, as this is the default alignment that we use in CHARONLANG. Given that this field is never used, it is also not declared, meaning that it could be represented as any type that fits into 4 bytes. In this case, this element's type is set to __unknown_type__ in the canonical representation, which occupies one word in the memory layout.

3.4.1 The Canonical Reconstruction. Given a certificate n, it is possible to obtain a canonical program P_c, such that $Cert_H(P_c) = n$. This reconstruction process is defined by a function $Canon(n)$, which Algorithm 1 shows. This reconstruction process happens in four stages:

(1) **Factorization:** Factorize the certificate n into a product of primes in the form p^{exp}, where p is the positional prime and *exp* is the construct-encoding exponent. This exponent is expected to follow the symbols and rules from Numbering Schema.

(2) **Analysis:** This step decides whether a data structure — i.e., a variable whose definition has more than one type symbol in *exp* — should be defined as an array or as a struct. This is achieved by analyzing all the use cases of these variables. If the variable's attributes are only accessed with static offsets, it will be defined as a struct; *Canon* defines it as an array otherwise.

(3) **Generation:** Iterate once more over the factorized certificate to produce the canonical program by matching the pattern from the Numbering Schema, and applying the rules discussed above.

(4) **Finalization:** As there is no explicit symbol for the main function in the certificate, and CharonLang assumes all the programs will follow a def-use relation, *Canon* will set the last function it defines to be the main function.

Algorithm 1 shows the pseudo-code of the reconstruction process for expressions, and it is implemented by the procedure *Canon*. As a simplification, it omits the handling of functions and control-flow constructs. The rest of this section illustrates how the reconstruction process in Algorithm 1 works through a series of examples, following Figure 17. This figure shows the canonical reconstruction of the certificate *n* from Figure 16.

```
2^((13)^(3)^(7)^(3)) *
3^((31)^(3)^(1)) *
5^((17)^(2)^(2)^(1)) *
7^((11)^(3)) *                    Ex. 3.9
11^(71) *
13^((17)^(2)^(2)^(3)) *
17^((11)^(4)) *
19^(71) *                         Ex. 3.9
23^((17)^(2)^(2)^(1)) *
29^((17)^(2)^(2)^(3)) *           Ex. 3.10
31^(79) *
37^(41) *
41^(37) *                         Ex. 3.11
43^(157)
```

```
struct struct_1 {
    int field_1;
    __unknown_type__ field_2;
    int field_3;
};

struct_1 var_1;

int main() {
    var_1.field_1 = 2;
    var_1.field_3 = 3;
    return var_1.field_1 + var_1.field_3;
}
```
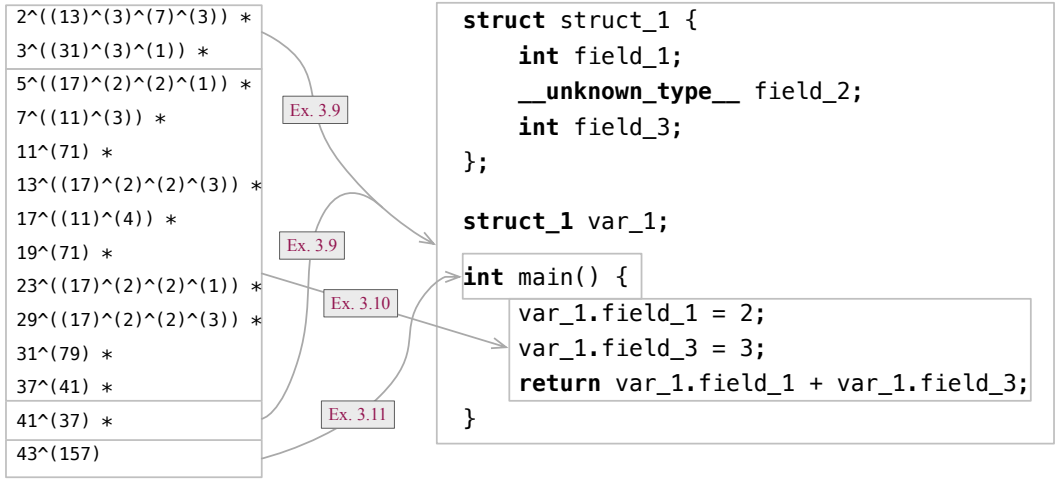
Fig. 17. The canonical reconstruction of a simple program.

Canon Parses Certificates Left-to-Right. The canonical program generation begins by parsing variable definitions ($exp. = 13^{var. \ def. \ exp.}$) and function parameters ($exp. = 19^{var. \ def. \ exp.}$). These are the first constructs encoded in the certificate, if any, as discussed in section 3.2. Variable definitions will be added to the program body immediately. Function parameters, on the other hand, are first added to the list of function parameters, and then passed to the program when *Canon* parses a Func. (start) symbol ($exp. = 31^{type \ symbol(type(f))^{(num.params.(f)+1)}}$). In this case, it will take the first n elements from this list and add to the function definition, where n is the number of parameters this function takes obtained from the symbol. *Canon* will name the n-th declared variable or parameter var_n, and the m-th defined function func_m, and will retrieve its type from *exp*. It will use these names to refer to the variable or parameter with the n-th variable prime, or the function with the m-th function prime in the certificate factors that follow.

Example 3.9. The certificate in Figure 17 encodes a single variable definition, `internal_var`. Its factor is $2\char`^((13)\char`^(3)\char`^(7)\char`^(3))$, and it tells this variable is a data structure as there are three type symbols in it (3, 7, and 3). *Canon* will search for variable use cases with 2 (the first prime) as the variable prime because this is the first variable to be declared. There are four instances: $5\char`^((17)\char`^(2)\char`^(2)\char`^(1))$, $13\char`^((17)\char`^(2)\char`^(2)\char`^(3))$, $23\char`^((17)\char`^(2)\char`^(2)\char`^(1))$, and $29\char`^((17)\char`^(2)\char`^(2)\char`^(3))$. The fields of the variable are accessed with a constant for index in all of the instances; as such, this variable will be set to be declared as a struct with an integer, an element of unknown type, and another integer — in this order. The next factor, $3\char`^((31)\char`^(3))$, marks the beginning of a function, which is first defined as `func_1`. This function scope will be closed when *Canon* reaches factor $41\char`^37$.

Expressions in Reverse Polish Notation. Expressions are parsed following a simple Reverse Polish Notation (RPN) inversion algorithm. Operands — variables ($exp. = 17^{var.\ usage\ exp.}$), constants ($exp. = 11^{c'}$), and function calls ($exp. = 29^{fp}$) — are always added to the expression stack. Function calls are also parsed following this notation: the algorithm will first parse the arguments ($exp. = 23$), and then the function call itself. When *Canon* finds a certificate factor that represents an operator, it will take the top elements (one if it is an unary operator, or two if it is a binary operator) from the stack, write the parenthesized expression, and add it back to the top of the stack. The expression is only added to the program body when the algorithm finds a terminator. Any factor that does not encode an operand or an operator is a terminator. The assignment operator is an exception for this rule, for it terminates an expression in RPN.

Example 3.10. Following, $5\char`^((17)\char`^(2)\char`^(2)\char`^(1))$ encodes a variable usage and causes *Canon* to start analyzing an expression. This certificate factor has 2 as the variable prime; as such, it refers to `var_1`. Given that this variable is a struct, and that the certificate factor encodes the access of the first element, *Canon* will add `.field_1` as element access suffix to the variable name it will push to the expression stack. $7\char`^((11)\char`^(3))$ comes right after it, and it encodes a constant. As the associated constant is positive, *Canon* will subtract 1 from it before pushing it to the expression stack. The next factor, $11\char`^(71)$, is the assignment operator, and the algorithm will write `var_1.field_1 = 2;`. As it is a terminator, this expression will be immediately added to `program`. This iterative process continues until *Canon* reaches the last factor, $43\char`^(157)$.

Reconstruction of Control Flow. Control-flow constructs are computed in *Canon* by first parsing the expression that follows a CοΝD factor ($exp. = 43$), until it finds the pattern that marks the beginning of a conditional code block (eg., IF (sτΑRτ), which has $exp = 47$). Then, it will add the control-flow construct followed by the conditional expression to the program body. Function and control-flow scopes are closed as soon as the algorithm finds the corresponding ending marker (eg., IF (εΝD), which has $exp. = 53$).

Example 3.11. As *Canon* finds the program-ending factor ($exp. = 157$), it will add the `main` function to the program it just created. It will rename the last function it defined — in this case, `func_1` — to `main`. Finally, it returns the canonical program P_c.

3.5 Correctness

This section provides sketches of proofs for the two main results discussed in this paper. In Section 3.5.1 we show that compilation preserves certificates. Then, in Section 3.5.2 we show that certificates are invertible.

3.5.1 Correctness of Equivalence of Certificates. To ensure that the certification process is correct, we establish that our compiler preserves the certification result. Specifically, for any high-level

Algorithm 1 CANON — The Reconstruction of Canonical Forms

1: **Input:** Certificate n
2: **Output:** Canonical program P_c
3:
4: **function** CANON(n)
5:     **return** CANONREC(F ← FACTORIZE(n), P ← empty program, S ← empty stack, v ← 0)
6:
7: **function** CANONREC(F, P, S, v)
8:     **if** F is empty **then**
9:         P.ADDMAIN()
10:        **return** P
11:     **end if**
12:     f, rest ← $F[0], F[1:]$
13:     e ← GETEXPFROMFACTOR(f)
14:     **if** ISVARDEF(e) **then**
15:        $v \leftarrow v + 1$
16:        name ← "var_" + v
17:        τ ← GETVARTYPE(e)
18:        **if** ISSINGLETYPE(τ) **then**
19:          P.ADDGLOBALVARIABLE(name, τ)
20:        **else if** ISSTRUCT(e) **then**
21:          fields ← GETFIELDSFROMVARTYPE(τ)
22:          P.ADDSTRUCT(name, fields)
23:        **else**
24:          size ← GETSIZEFROMVARTYPE(τ)
25:          P.ADDARRAY(name, τ, size)
26:        **end if**
27:     **else if** ISVARUSAGE(e) **then**
28:        name ← GETVARNAMEFROMVARUSAGE(e)
29:        S.PUSH(name)
30:     **else if** ISCONSTANT(e) **then**
31:        S.PUSH($(c - 1)$ if $c \geq 0$ else c)
32:     **else if** ISOPERATION(e) **then**
33:        op ← GETOPERATIONSYMBOLFROMEXP(e)
34:        rhs ← S.POP()
35:        **if** ISUNARY(e) **then**
36:          expr ← (op + rhs)
37:        **else**
38:          lhs ← S.POP()
39:          expr ← (lhs + op + rhs)
40:        **end if**
41:        S.PUSH(expr)
42:     **end if**
43:     **if** ISTERMINATOR(e) **then**
44:        stmt ← S.POP()
45:        P.ADDEXPR(stmt)
46:     **end if**
47:     **return** CANONREC(rest, P, S, v)

program P_H compiled to a low-level program P_L, the outputs of the certification algorithms $Cert_H$ and $Cert_L$ coincide. This property requires the n-th variable prime vp_n produced by both certification

algorithms $Cert_H$ and $Cert_L$ to be equal. This is proven in Lemma 3.12. It shows that the certification environments C_H and C_L produced for the high- and low-level certification algorithms, respectively, map the high- and low-level representations of a left-hand side term a to the same variable prime.

LEMMA 3.12 (EQUIVALENCE OF VARIABLE PRIMES). *If a is a left-hand side term, and $Comp(a) = \ldots T(a)$, then $C_H(a) = C_L(T(a))$.*

PROOF. The proof goes by case analysis on the rules shown in Figure 10. Consider the case where $a = v$, i.e., a variable. The corresponding rule Figure 10 shows that if $v \notin C_H$, then a fresh variable prime $vp' = \text{next}(vp)$ is generated and C_H is updated with $v \mapsto vp'$, while C_L is updated with $T(v) \mapsto vp'$. Thus, both environments remain equivalent, assigning the same prime to v and to its corresponding address in the low-level domain.

The array access case, $a = v[e]$, happens in two steps. First, v is compiled under the same rule as above, assigning it a prime if it is not already mapped to one in C_H and C_L. Then, the index e is compiled separately. If e happens to be a variable, then it will also be subject to the *variable* rule, and the certification environments will be updated accordingly. If not, then the compilation of e will not change C_H and C_L. As such, any changes to the certification environment will be consistent across C_H and C_L.

For $a = v.x$, i.e., a field access in a struct, the rule shows that the base variable v is compiled following the first rule, and then the offset $T(v.x)$ is added to it. Since C_H and C_L are only updated when v is compiled (and not for the field access itself), consistency is preserved for the assigned variable prime.

Finally, in the case of function definitions, $\theta_f\ f$, the rule specifies that if $f \notin C_H$, a fresh function prime $fp = \text{next}(fp)$ is generated. The environments C_H and C_L are then updated with $f \mapsto fp$ and $T(f) \mapsto fp$, respectively. Since the same prime is used in both updates, the environments remain equivalent.

In all cases, when a fresh prime is introduced, it is added to both environments simultaneously. Thus, we have that for all a, the relation $C_H(a) = C_L(T(a))$ is true. □

Theorem 3.13 is the core result of this work. It establishes that certification is preserved through compilation: the certificate computed from a high-level program matches exactly the one computed from its compiled low-level form. This result ensures that certificates abstract program behavior in a way that is invariant under compilation.

THEOREM 3.13 (PRESERVATION OF CERTIFICATES). *If P_H is a program in the high-level CHARONLANG language, then $Cert_H(P_H) = Cert_L(Comp(P_H))$.*

PROOF. The proof proceeds by structural induction on the syntax of the source program P_H.

Base cases: the certification of constants, variables, array and struct accesses does not involve the recursive certification of subexpressions; thus, the theorem follows without induction:

Constant: If c is a constant, then $Cert_H(c) = Cert_L(\text{CONSTANT } R_c \text{ c}) = 2^{11^c}$

Variable: If x is a variable name, then $Cert_H(x) = Cert_L(\text{CONSTANT } R_c \text{ x}; \text{ ADD } R_x \text{ } R_c \text{ zero}) = p^{17^{var.\ usage\ exp.}}$. The exponent *var. usage exp.* tells whether this is a simple variable or an element in a data structure — and whether it is accessed with a static [5] or a dynamic offset[6]. If it is a data structure, the high- and low-level programs P_H and P_L programs will also contain expressions or instructions to calculate the element offset from the base variable. In either case, $Cert_H$ and $Cert_L$ will have the same environments C_H and C_L, as a direct result from Lemma 3.12. As *var. usage exp.* depends solely on the expressions and instructions

---

[5]As an array with constant for index or element in a struct

[6]As an array with a variable for index

that implement the variable and the certification environments C_H and C_L, we can conclude that the high- and low-level certification of variables coincide.

Inductive step: We apply induction to prove preservation of certificates on constructs whose syntax is defined recursively in Figure 5. Thus, if we have a construct defined as $S_0 ::= \ldots S_1 \ldots$, then we assume preservation over S_1 to demonstrate that it holds for S_0. As an example, we shall consider the compound program construct $P_H = S_0 = \text{if}(e) \{S\}$. According to the compilation rules (Fig. 8), the compiler emits the following instruction sequence for such conditionals:

$$Comp(\text{if}(e) \{S\}) = I_e;\ \text{JZ } R_e\ \ell;\ I_t,\ T'$$
$$Comp(e) \rightarrow I_e,\ R_e$$
$$Comp(S) \rightarrow I_t,\ T'$$

From the certificate construction rules (Fig. 13), we know:

$$Cert_H(\text{if}(e) \{S\}) = p^{43} \times Cert_H(e) \times next(p)^{47} \times Cert_H(S) \times next(p)^{53}$$

Similarly, from the low-level certificate construction rules (Fig. 14):

$$Cert_L(I_e;\ \text{JZ } R_e\ \ell;\ I_t) = p^{43} \times Cert_L(I_e) \times next(p)^{47} \times Cert_L(I_t) \times next(p)^{53}$$

By the inductive hypothesis, we assume:

$$Comp(e) = I_e \Rightarrow Cert_H(e) = Cert_L(I_e)$$
$$Comp(S) = I_t \Rightarrow Cert_H(S) = Cert_L(I_t)$$

We now observe that the same structure appears in both $Cert_H$ and $Cert_L$. From this, we can conclude that the conditional symbol p^{43} and the the control flow symbols $next(p)^{47}$ and $next(p)^{53}$ will be placed identically in both constructions, as they depend exclusively on the structure of the programs. Therefore,

$$Cert_H(\text{if}(e) \{S\}) = Cert_L(Comp(\text{if}(e) \{S\}))$$

The proof of equivalence between $Cert_H$ and $Cert_L$ for other constructs is analogous.   □

Unlike the certification of high-level programs, where $Cert_H(P_{H_1})$ might be equal to $Cert_H(P_{H_2})$ even if $P_{H_1} \neq P_{H_2}$, $Cert_L(P_{L_1})$ will be equal to $Cert_L(P_{L_2})$ if, and only if $P_{L_1} = P_{L_2}$. This property is demonstrated in Theorem 3.14.

THEOREM 3.14 (UNIQUENESS OF LOW-LEVEL PROGRAMS). *Let P_1 and P_2 be two programs in the low-level CHARONIR intermediate representation. Then $Cert_L(P_1) = Cert_L(P_2)$ if and only if $P_1 = P_2$.*

PROOF. The *if* case, $Cert_L(P_1) = Cert_L(P_2)$ if $P_1 = P_2$, is trivial because $Cert_L$ is deterministic. Therefore, passing the same certificate to $Cert$ yields the same result.

To prove $Cert_L(P_1) = Cert_L(P_2)$ only if $P_1 = P_2$, we'll use structural induction on the certification rules for $Cert_L$ from Figure 14.

Base case: The certification of constants, variables definitions, and variable usage follows patterns with a fixed set of instructions. As such, the equality of these constructs in programs P_1 and P_2 and can be demonstrated without induction.

   Constant: The certificate encodes the constant c from the instruction CONSTANT R_c c in the exponent, following the formula $11^{c+1\ if\ c \geq 0\ else\ c}$. As this formula is injective, the equality of certificates implies the same CONSTANT instruction appears in the same position in both programs.

Variable definition: A variable definition is encoded from an address in data section of a program. It is obtained from $13^{type\ symbol(var_{address})}$, where *type symbol* encodes the inferred type of this variable. If both programs have the same certificate, then the same type has been inferred for the same address of data. This implies programs P_1 and P_2 declare the exact same variables.

Variable usage: The variable usage certificate follows the formula $p^{17^{var.\ usage.exp.}}$. The exponent, *var. usage exp.*, will vary depending on the variable usage construct it encodes:

Scalar: A scalar is implemented in low-level from a sequence of instructions that will load the variable base address into a temporary register. In this case, *var. usage exp.* = $C_L(x)^2$, x is the variable's base memory address and $C_L(x)$ is its variable prime in the certification environment.

Variable usage with static offset: The implementation for this construct begins similarly as the previous construct, loading the base memory address into a temporary register. However, it will use additional instructions to load the constant offset into another temporary register, and add the contents of the former and the latter registers into a third register, that will contain the address of interest. The exponent will be *var. usage exp.* = $C_L(x)^{2^{(1+offset)}}$, where *offset* is the constant offset and x and $C_L(x)$ follow the same previous notation.

Variable usage with dynamic offset: The last case for the variable usage construct also starts by loading the base memory address into a temporary register. Unlike the previous case, it will load the contents of another variable into another temporary register, and have memory address arithmetic instructions to compute the address of interest in runtime. Its exponent will be *var. usage exp.* = $C_L(x)^{3^{C_L(y)}}$, where $C_L(x)$ and $C_L(y)$ are the variable primes of the variable being used and of the variable to dynamically compute the memory offset with. x is the base memory address.

If the certificates are identical, then the same sequence of instructions must be present in the same position in both programs, the variables must use the same addresses, and any offsets from the base memory addresses must be identical, in either variable construct the programs implement. Otherwise, there would be at least one differing variable usage certificate.

Inductive step: The inductive step applies onto recursive syntactic constructions. We shall analyze the if-then (no else) case. The other control-flow cases, if-then-else and while, are similar. Assume that the equality holds for all the pairs of instructions sequences (P_{1_1}, P_{2_1}), (P_{1_2}, P_{2_2}), ... Consider the certification of "*if*(S){S'}", $p^{43} \times Cert_L(S) \times next(p)^{47} \times Cert_L(S') \times next(p)^{53}$.

If this construct is encoded in the certificate, then p^{43} will mark the beginning of the predicate in the same position in both programs. From the inductive hypothesis, the next element, $Cert_L(S)$, will tell the exact same sequence of instructions S implement the predicate. The hypothesis also guarantees that the next three elements, $next(p)^{47} \times Cert_L(S') \times next(p)^{53}$, will encode the same sequence S' preceded by the same conditional jump in the same positions in both programs. Therefore:

$$Cert_L(P_1) = Cert_L(P_2) \iff P_1 = P_2$$

□

3.5.2 Correctness of Invertibility. The canonical form is constructed from the certificate produced by either $Cert_H$ or $Cert_L$, as discussed in Section 3.4. The program produced by *Canon* captures the logical structure implied by the certificate, independent of the specific syntactic choices made in

the source or compiled program. If two programs yield the same certificate, then they must share the same canonical form. This property is demonstrated in Theorem 3.15.

THEOREM 3.15 (UNIQUENESS OF CANONICAL FORM). *Let P_1 and P_2 be two programs in the high-level CHARONLANG representation. Then $Canon(Cert_H(P_1)) = Canon(Cert_H(P_2))$ if and only if $Cert_H(P_1) = Cert_H(P_2)$.*

PROOF. The proof for both directions is presented below:

$\Leftarrow$ If $Cert_H(P_1) = Cert_H(P_2)$, then $Canon(Cert(P_1)) = Canon(Cert(P_2))$.

The proof follows directly from the determinism of the *Canon* algorithm (Algorithm 1). As the inputs are identical, then it will produce the same result in both cases.

$\Rightarrow$ If $Canon(Cert(P_1)) = Canon(Cert(P_2))$, then $Cert_H(P_1) = Cert_H(P_2)$.

Let $Cert_H(P_1) = n_1$ and $Cert_H(P_2) = n_2$, and $n_1 = f_1 \times f_2 \times \ldots$ and $n_2 = f_1' \times f_2' \times \ldots$. We have that $F = [f_1, f_2, \ldots,]$ and $F' = [f_1', f_2', \ldots,]$, where F (and F') is the list of factors that CANON (Algorithm 1) passes to CANONREC as the first argument. The proof follows by induction on the length of F. We shall demonstrate that $F = F'$ so that these two invocations of CANONREC produce the same program. On the base case, we have that $F = [] = F'$. Thus, CANONREC$([], empty\ program, [], 0) = $ CANONREC$([], empty\ program, [], 0)$, and it follows that *empty program = empty program*.

On the inductive case, we assume that the theorem holds up to a list of factors of length k. Thus, we need to show that CANONREC$([f_k, \ldots], P_k, S_k, v_k) = $ CANONREC$([f_k', \ldots], P_k', S_k', v_k')$. We assume, by induction, that $P_k = P_k'$, $S_k = S_k'$ and $v_k = v_k'$. In what follows, we analyze some of the cases that Algorithm 1 handles.

Constant: Assume CANONREC$([f, \ldots], P, S, v) = $ CANONREC$([f', \ldots], P, S, v)$. If the factor f is the certificate of a constant, then it has the exponent $e = 11^c$. The only way the equality can hold is if the factor f' has the exponent $e' = 11^{c'}$, where $c = c'$, following **else if** ISCONSTANT.

Variable definition: Assume CANONREC$([f, \ldots], P, S, v) = $ CANONREC$([f', \ldots], P, S, v)$. If f encodes the definition of a variable, it has an exponent in the form $e = 13^{var.\ def.\ exp.}$. The equality will only hold if f' also encodes a variable definition with an exponent $e' = 13^{var.\ def.\ exp.'}$, so CANONREC will parse f and f' with the case **if** ISVARDEF. Additionally, it is necessary that *var. def. exp. = var. def. exp.'* for the types τ and τ' the algorithm extracts with GETVARTYPE to be identical. The equality between τ and τ' is required for CANONREC to add the same global variable, or struct with the same fields, or array with the same length to the canonical program being reconstructed.

Variable usage: Assume CANONREC$([f, \ldots], P, S, v) = $ CANONREC$([f', \ldots], P, S, v)$. If f is a factor that encodes the usage of a variable, then its exponent must be $e = 17^{var.\ usage\ exp.}$. For the equality to hold, f' must also have a variable usage encoding exponent, $e' = 17^{var.\ usage\ exp.'}$, so it will fall into the same case — **else if** ISVARUSAGE. In particular, f and f' must encode the usage of the same variable, such that *var. usage exp. = var. usage exp.'*, and the same variable prime vp can be retrieved from it. This is mandatory for CANONREC to obtain the same variable name from GETVARNAMEFROMVARUSAGE — this method will return the name var_n, where n is the index of the variable prime vp in the ordered set of prime numbers.

Operation: Assume CANONREC$([f, \ldots], P, S, v) = $ CANONREC$([f', \ldots], P, S, v)$. If f is the certificate of an operation, then CANONREC will parse it with the **else if** ISOPERATION case. It is required that f and f' encode the same operation for the equality to hold, so GETOPERATIONFROMEXP will return the same symbol in both cases.

Thus, we have that if $\text{CanonRec}(F, \textit{empty program}, [], 0) = \text{CanonRec}(F', \textit{empty program}, [], 0)$, then $F = F'$, and therefore, $Cert_H(P_1) = Cert_H(P_2)$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

The canonical form is also preserved by the compilation process. In other words, if two programs P_1 and P_2 share the same canonical form, then they compile to the same low-level program P_L. Theorem 3.16 proves this property.

THEOREM 3.16 (PRESERVATION OF CANONICAL FORMS). *Let P_1 and P_2 be two programs such that $Canon(Cert_H(P_1)) = Canon(Cert_H(P_2))$. In this case, $Comp(P_1) = Comp(P_2)$.*

PROOF. Theorem 3.15 demonstrates that if two programs P_1 and P_2 share the same canonical form, then they must have the same certificate. From this property, we conclude that $Cert_H(P_1) = Cert_H(P_2)$ because $Canon(Cert_H(P_1)) = Canon(Cert_H(P_2))$, from the hypothesis.

Theorem 3.14, on the other hand, shows that two programs with the same certificate will compile to the same low-level program. As we've learned that $Cert_H(P_1) = Cert_H(P_2)$ from the previous step, then $Comp(P_1) = Comp(P_2)$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

4 Evaluation

This section explores the following research questions:

(1) **RQ1:** How does the certificate length scale with the size of the high- and low-level source programs (AST nodes and instructions)?

(2) **RQ2:** How does the running time of the high- and low-level certification algorithms grow with program size?

Hardware/Software. Experiments evaluated in this section were performed on an Apple M1 CPU, featuring 8 GB of integrated RAM, clock of 3.2 GHz, and banks of 192 KB and 128 KB L1 caches — four of each. The experimental setup runs on macOS Sequoia 15.4.1. The CHARON project is implemented in Python version 3.12.4 (Jun'24).

Benchmarks. The GitHub repository of the CHARON project[7] contains 20 programs written in CHARONLANG. Each program exercises different features of the high-level language. These programs vary in size, which can be measured in terms of the number of statements, AST nodes, and instructions. This diversity allows us to evaluate the asymptotic behavior of the certification technique proposed in this paper. The corresponding results are summarized in Figures 18 and 19, and will be discussed in the remainder of this section.

4.1 RQ1: On the Size of Certificates

The compilation certificate is represented as a number whose magnitude grows exponentially with the quantity of constructs in the target program, since each additional construct introduces a new multiplicative factor. However, the certification process does not require computing or storing the exact value of this number. Instead, the certificate is maintained in its symbolic form; that is, as a string representing the sequence of multiplicative expressions used to construct it. Therefore, we measure the size of a certificate based on its symbolic representation, defined as the character length of its string form, rather than by the numeric magnitude of the final value. In this section, we show that although the absolute value of a certificate grows exponentially, its symbolic representation grows linearly with the size of the program.
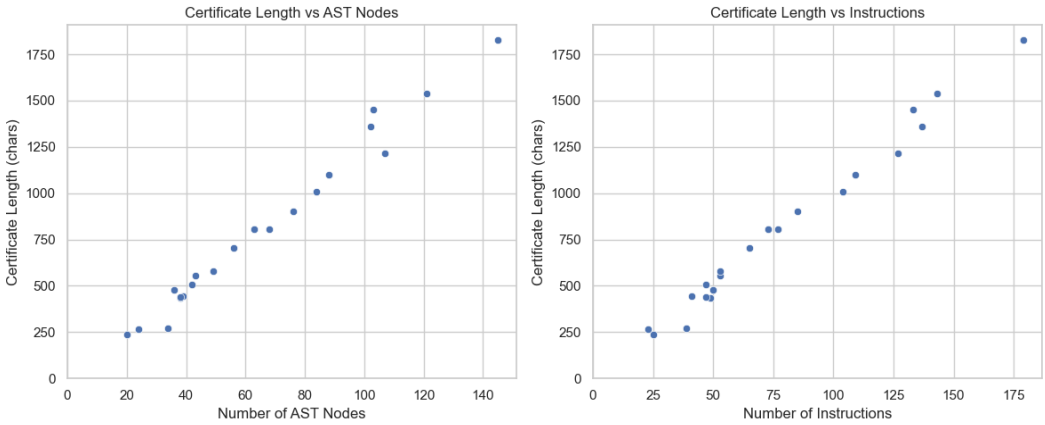
---

[7]https://github.com/guilhermeolivsilva/project-charon

Fig. 18. Certificate expression length (in characters) vs program size.

Discussion. Figure 18 illustrates the relationship between program size—measured by the number of AST nodes—and certificate length. The size of the certificates grows linearly with the size of the input program, both in terms of high-level AST nodes and low-level instructions.

In both cases, the linear correlation is very strong: the Pearson correlation coefficient (R^2) between the number of AST nodes and the certificate length is 0.98. Similarly, the R^2 value between the length of certificates and the number of instructions in the CHARONIR representation is 0.97. This linear trend reflects the design of the certification rules, where each program construct contributes a constant-size component to the overall certificate expression.

4.2 RQ2: On the Performance of the Certifier

We evaluate algorithmic performance by measuring the average running time of the certification procedures across a suite of test programs, using Python's `%%timeit` for high-resolution timing. Our current implementation, written in Python, is not optimized for performance and inherits the inefficiencies associated with the language. However, in this section, we demonstrate that the asymptotic complexity of the certification process grows linearly with the size of the input program. This suggests that a reimplementation in a systems-oriented language such as C, C++ or Rust, would likely result in significantly faster certification times.

Discussion. Figure 19 presents the correlation between the running time of the certification process and the size of the certified programs. The left-hand plot shows the performance of the high-level algorithm (as a function of AST nodes), and the right-hand plot analyzes the low-level algorithm (as a function of instruction count). The high-level certification algorithm runs in sub-millisecond time across the entire benchmark suite. As shown in Figure 19 (left), the runtime scales linearly with the number of AST nodes, reaching approximately 1.6 ms for the largest inputs, which contains 145 nodes. The coefficient of correlation between running time and size is 0.96; hence, very close to a perfectly linear correlation. This behavior confirms the expected complexity of the algorithm, which performs a recursive descent over the abstract syntax tree with constant-time processing at each node.

The low-level certification algorithm, shown in Figure 19 (right), exhibits even faster execution times, typically in the tens of microseconds. It processes the instruction stream through a sequential scan based on pattern matching. Although the low-level certification procedure demonstrates
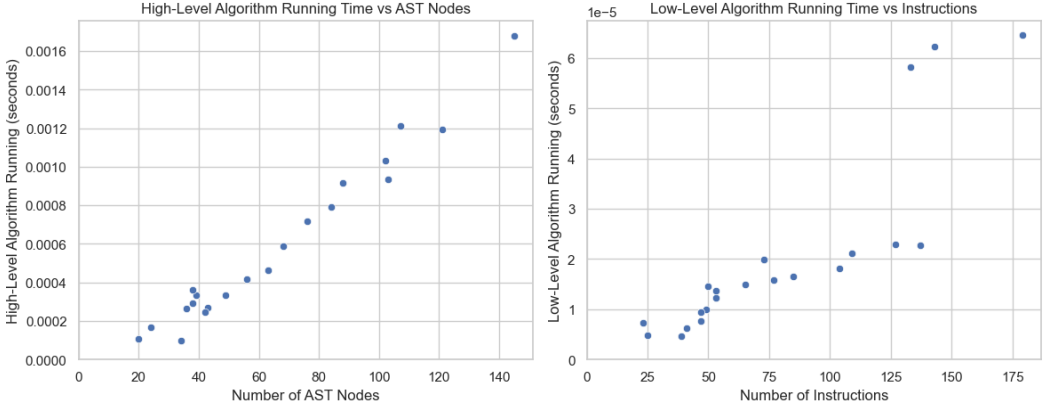
Fig. 19. Running time (in seconds) vs program size.

near-linear behavior in practice ($R^2 = 0.93$), it is not guaranteed to run in linear time. In particular, the algorithm may exhibit quadratic behavior in the presence of deeply nested branches. This super-linear behavior is observable in our results: the three largest programs in the benchmark suite deviate from the linear trend. These outliers contain several control-flow constructs (e.g., `if` and `while` statements), which trigger multiple executions of the *Analyze-Conditional-Jumps* pass.

5 Related Work

This paper introduces what we call "*Structural Certification*", a certification methodology in which the layout of the binary itself encodes evidence of its derivation from the source code. We believe this perspective is novel when compared to prior work. Nevertheless, as mentioned in Section 1, there exists a substantial body of research that shares our goal: building trust between users and compilers. In the remainder of this section, we review several techniques aligned with this objective.

5.1 Diverse Double-Compiling

Diverse Double-Compiling (DDC) [Wheeler 2005, 2010], along with its many adaptations [Drexel et al. 2025; Höller et al. 2015; Skrimstad 2018; Somlo 2020], is a defense technique proposed by Wheeler in his PhD dissertation. It seeks to detect malicious behavior introduced by a compromised compiler by comparing the outputs of independently built compiler binaries. The method proceeds in four steps:

(1) Compile the source code of the compiler C under test using a trusted compiler C_t, producing an executable compiler $X_t = C_t(C)$.
(2) Compile the same source using the suspect compiler C_s, producing an executable compiler $X_s = C_s(C)$.
(3) Use both resulting compiler executables to compile a test program p.
(4) Compare the resulting binaries, i.e., $b_t = X_t(p)$ and $b_s = X_s(p)$.

The key insight is that a trusted compiler C_t, assumed to be free of malicious logic, should produce a clean version X_t of the compiler C. If the executables X_t and X_s produce different binaries in Step 3, this discrepancy can be detected through syntactic comparison in Step 4.

DDC serves as a practical defense against self-propagating attacks such as Thompson's "Trusting Trust" backdoor. However, it relies on two assumptions [Carnavalet and Mannan 2014]: (i) the existence of at least one trusted compiler C_t, and (ii) that both X_t and X_s produce deterministic

outputs for the same input. Furthermore, any differences observed in Step 4, such as $b_t \neq b_s$, require additional investigation to determine whether they stem from benign implementation differences or intentional tampering.

5.2 Lightweight Certification Approaches

Lightweight certification techniques, such as Proof-Carrying Code (PCC) [Necula 1997] and Typed Assembly Language (TAL) [Morrisett et al. 1999], enable validation of binary code without requiring trust in the compiler. They do so by embedding verifiable metadata either alongside or within the compiled program. In PCC, the compiler generates a logical proof that the binary satisfies a predefined safety policy; this proof is distributed with the code and checked by the consumer prior to execution. TAL follows a similar approach, embedding type information directly in the assembly code to guarantee properties such as control-flow correctness and memory safety.

Our work shares the same high-level goal; that is, allowing untrusted code to be verified independently; however, it adopts a different methodology. Rather than attaching external proofs or annotations to the code, we encode a certificate of correct compilation directly into the structure of the binary, using a Gödel-style numbering schema. This certificate is not an auxiliary artifact. It is instead embedded into the form of the binary code itself, enabling verification through basic arithmetic checks. In contrast to PCC and TAL, which require logical or type-based reasoning engines, our approach supports self-contained, solver-free certification by leveraging the regularity and reversibility of the compilation process.

5.3 Symbolic Certification

Symbolic certification [Barthe et al. 2005; Govereau 2012; Kirchner et al. 2005; Rival 2004; Zaks and Pnueli 2008] refers to techniques that verify the correctness of compiled code by constructing and comparing symbolic representations of program semantics at both the source and binary levels, ensuring that they exhibit behaviorally equivalent execution. While sharing a similar goal to our approach, symbolic certification is technically distinct and offers a different set of guarantees. We illustrate this distinction by analyzing the work of Rival [2004].

In Rival's framework, the correctness of compilation is established through the construction of *symbolic transfer functions* (STFs), which encode the semantics of source and binary code in a compositional manner. Correctness is then verified by demonstrating that these symbolic descriptions are behaviorally equivalent. This approach has two advantages over ours: (i) it supports optimized compiler backends, and (ii) it enables the verification of semantic properties such as the absence of runtime errors or the preservation of invariants.

By contrast, our technique encodes a certificate of correct compilation directly within the binary. Instead of relying on symbolic reasoning or logical inference, we use a numeric encoding schema that maps constructs in the source program to patterns in the compiled binary. This yields a form of *certification by structure*, which, unlike symbolic certification, requires no external proofs, solvers, or trusted verifiers. Although symbolic techniques like that of Rival offer broader expressiveness and can validate aggressive compiler optimizations, they still ask for trust in the verification infrastructure. In contrast, our method enables a self-contained form of certification, where correctness can be checked arithmetically from the structure of the program alone.

6 Conclusion

This paper has introduced a new certification methodology, which we call *structural certification*. In this approach, the certificate of correct compilation is inherently embedded in the structure of the binary code itself. Unlike previous techniques, such as proof-carrying code, translation validation, or verified compilation, which ask for trust in external artifacts, our approach enables

certification through the form and layout of the compiled program. Inspired by Gödel's numbering system, our method encodes syntactic and semantic properties of the source program directly into the binary using arithmetic encodings derived from prime factorizations. This design enables lightweight verification that requires neither theorem provers nor trusted third-party verifiers. We have demonstrated the feasibility of this idea through the development of Charon, a compiler for a subset of C capable of certifying programs written in FaCT. By embedding certification directly into the binary representation, our method opens a new path for certifying compilers—one in which the evidence of correct compilation is inseparable from the code it certifies.

Limitations. While structural certification offers a novel and lightweight approach to trusted compilation, it comes with limitations. Most notably, the current method assumes a non-optimizing, syntax-directed compiler, where each source construct maps deterministically to a corresponding binary pattern. This limitation restricts support for aggressive low-level optimizations, such as instruction reordering or register allocation. These transformations, if applied on the low-level code would invalidate the numeric correspondence required by our encoding. Thus, programs certified via our approach must be optimized at the source-code level. Moreover, because the certification relies on the preservation of positional and syntactic structure, it may not scale naturally to full-featured programming languages or to target architectures with highly complex instruction sets. Finally, our method verifies structural fidelity between source and binary, but does not presently capture deeper semantic properties, such as memory safety or functional correctness.

Future Work. These limitations suggest several directions for future research. One avenue is to extend the structural encoding to tolerate limited classes of compiler optimizations, perhaps by generalizing the encoding to sets of acceptable patterns or by embedding structured metadata alongside numeric codes. Another possibility is to enrich the certification mechanism to express and verify semantic properties, such as safety or constant-time guarantees, while retaining the lightweight, arithmetic-checkable format. Additionally, it would be valuable to explore how structural certification interacts with techniques like diverse double-compiling or translation validation, potentially combining their strengths to build higher-assurance compilation pipelines. Finally, further work is needed to assess the robustness of structural certification in adversarial settings, where attackers may attempt to tamper with or forge encodings in compiled binaries.

Software. The certifying infrastructure described in this paper is available at https://github.com/guilhermeolivsilva/project-charon.

On the Usage of AI-Assisted Technologies in the Writing Process

During the preparation of this manuscript, the authors used ChatGPT (version 3.5), a language model developed by OpenAI, to revise grammar and style in portions of the text. This tool was used on June of 2025, and all AI-assisted content was subsequently reviewed and edited by the authors. The authors accept full responsibility for the content and conclusions presented in this publication.

References

Krste Asanović and David A. Patterson. 2014. Instruction Sets Should Be Free: The Case For RISC-V. Technical Report UCB/EECS-2014-146. University of California at Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

Gilles Barthe, Tamara Rezk, and Ando Saabas. 2005. Proof obligations preserving compilation. In International Workshop on Formal Aspects in Security and Trust. Springer, 112–126.

Xavier Carnavalet and Mohammad Mannan. 2014. Challenges and implications of verifiable builds for security-critical open-source software. In Computer Security Applications Conference. 16–25.

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In PLDI (Phoenix, AZ, USA). Association for Computing Machinery, New York, NY, USA, 174–189. https://doi.org/10.1145/3314221.3314605

Carmine Cesarano, Vivi Andersson, Roberto Natella, and Martin Monperrus. 2024. GoSurf: Identifying Software Supply Chain Attack Vectors in Go. In SCORED (Salt Lake City, UT, USA). Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/3689944.3696166

Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58, 2 (1936), 345–363.

Russ Cox. 2023. Running the "Reflections on Trusting Trust" Compiler. https://research.swtch.com/nih. Accessed: 2024-12-30.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

N.G. de Bruijn. 1994. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem**Reprinted from: Indagationes Math, 34, 5, p. 381-392, by courtesy of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam. In Selected Papers on Automath, R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 133. Elsevier, 375–388. https://doi.org/10.1016/S0049-237X(08)70216-7

Joshua Drexel, Esther Hänggi, and Iyán Méndez Veiga. 2025. Reproducible Builds and Insights from an Independent Verifier for Arch Linux. arXiv preprint arXiv:2505.21642 (2025).

Carl Friedrich Gauss. 1966. Disquisitiones arithmeticae. Yale University Press.

Kurt Gödel. 1931. über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik 38, 1 (1931), 173–198. https://doi.org/10.1007/BF01700692 Translated title: "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I".

Paul Govereau. 2012. Denotational Translation Validation. Ph. D. Dissertation.

Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner. 2015. Evaluation of diverse compiling for software-fault detection. In DATE. IEEE, 531–536.

Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. 2005. Formal validation of pattern matching code. In PPDP. Association for Computing Machinery, New York, NY, USA, 187–197.

Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, and Olivier Barais. 2023. Journey to the Center of Software Supply Chain Attacks. IEEE Security and Privacy 21, 6 (Aug. 2023), 34–49. https://doi.org/10.1109/MSEC.2023.3302066

Xavier Leroy. 2009. Formal verification of a realistic compiler. Commun. ACM 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. ACM Trans. Program. Lang. Syst. 21, 3 (May 1999), 527–568. https://doi.org/10.1145/319301.319345

George C. Necula. 1997. Proof-carrying code. In POPL (Paris, France). Association for Computing Machinery, New York, NY, USA, 106–119. https://doi.org/10.1145/263699.263712

George C. Necula. 2000. Translation validation for an optimizing compiler. In PLDI (Vancouver, British Columbia, Canada). Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In DIMVA (Lisbon, Portugal). Springer-Verlag, Berlin, Heidelberg, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2

Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. 2024. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. arXiv:2406.10109 [cs.CR] https://arxiv.org/abs/2406.10109

Amir Pnueli, Ofer Shtrichman, and Michael Siegel. 1999. Translation Validation: From SIGNAL to C. In Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel). Springer-Verlag, 231–255.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In TACAS. Springer-Verlag, Berlin, Heidelberg, 151–166.

Silvain Rideau and Xavier Leroy. 2010. Validating register allocation and spilling. In ETAPS (Paphos, Cyprus). Springer-Verlag, Berlin, Heidelberg, 224–243. https://doi.org/10.1007/978-3-642-11970-5_13

Xavier Rival. 2004. Symbolic transfer function-based approaches to certified compilation. In POPL (Venice, Italy). Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/964001.964002

Yrjan Skrimstad. 2018. Improving trust in software through diverse double-compiling and reproducible builds. Master's thesis.

Gabriel L Somlo. 2020. Toward a Trustable, Self-Hosting Computer System. In SPW. IEEE, 136–143.

Ken Thompson. 1984. Reflections on trusting trust. Commun. ACM 27, 8 (Aug. 1984), 761–763. https://doi.org/10.1145/358198.358210

Alan M. Turing. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 2, 42 (1936), 230–265.

David Wheeler. 2005. Countering Trusting Trust through Diverse Double-Compiling. In ACSAC. IEEE Computer Society, USA, 33–48. https://doi.org/10.1109/CSAC.2005.17

David A. Wheeler. 2010. Fully Countering Trusting Trust through Diverse Double-Compiling. arXiv:1004.5534 [cs.CR] https://arxiv.org/abs/1004.5534

Anna Zaks and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In International Symposium on Formal Methods. Springer, 35–51.