# zkTAL: Type Checking Assembly in Zero-Knowledge

Luís Ferreirinha
*Vrije Universiteit Amsterdam*
Amsterdam, the Netherlands
l.p.felixferreirinha@vu.nl

Klaus v. Gleissenthall
*Vrije Universiteit Amsterdam*
Amsterdam, the Netherlands
k.freiherrvongleissenthal@vu.nl

***Abstract*—In this short paper, we introduce zkTAL, a novel approach that encodes type derivations of typed assembly language programs within a zero-knowledge proof system. Our method allows end-users to efficiently verify that a given binary adheres to a specified type system, preserving memory and control-flow safety guarantees, without revealing sensitive information. As a preliminary demonstration, we implement a proof-of-concept using Noir, a zkSNARK framework, to verify that programs for a simple typed assembly language, with control-flow safety, were correctly type-checked in zero-knowledge.**

***Index Terms*—typed assembly language, type checker, formal verification, zero-knowledege, zksnark**

## I. Introduction

Proprietary applications, such as Microsoft Office, Adobe Creative Suite, Windows, and proprietary device drivers and firmware, are among the most widely used software today. Unlike their open-source counterparts, their source code is inaccessible to public audit and available only to company developers, requiring users to put their trust in the vendors. However, these proprietary applications are frequently found to contain vulnerabilities that compromise user data and system integrity [1]. Unfortunately, due to their closed-source nature, users cannot independently verify whether the software they rely on is functionally correct or free from vulnerabilities.

This leads to the question: How can end-users ensure that the software they are executing is safe? Currently, they are faced with only two choices: trust the software provider or attempt to reverse engineer the program to uncover potential weaknesses. However, the latter is impractical, as reverse engineering a binary is an enormous undertaking, especially since software vendors commonly obfuscate or strip binaries to protect intellectual property (IP), greatly complicating such efforts. The alternative would be to have vendors directly provide a proof for users to independently verify that their software is indeed safe. However, this would inadvertently reveal sensitive information about the program design, thus compromising IP.

To tackle this issue, we propose zkTAL, an approach that allows vendors to preserve their IP while allowing end-users to independently verify the safety of the software they execute. We realize this approach, by combining formal verification with cryptographic zkSNARK proofs. Among formal verification techniques, type systems stand out as the most lightweight and widely adopted. For example, the Rust programming language employs a robust type system that enables developers to write memory-safe programs [2]. Inspired by this, we develop a type system for RISC-V assembly, that enables formal proofs showing that a binary annotated with type information is free of memory errors, if it type-checks. To address the concern of information leakage, the vendor must supply a cryptographic proof stating compliance with the type system, rather than shipping the proof directly. Such a cryptographic proof must simultaneously conceal the actual types and be efficiently verifiable by the end-user. Zero-knowledge succinct arguments of knowledge (zkSNARKs) [3] effectively meet these requirements, as they provide zero-knowledge assurances and efficient verification. In this short paper, we present zkTAL a novel approach to encoding correctness proofs for typed assembly programs within a zero-knowledge proof system, thereby enabling the end-user to efficiently verify that the program adheres to the type system without compromising sensitive information.

## II. Overview

The verification of a binary using zkTAL consists of two phases: proof generation by the vendor, and proof verification by the user. Figure 1 illustrates this process.

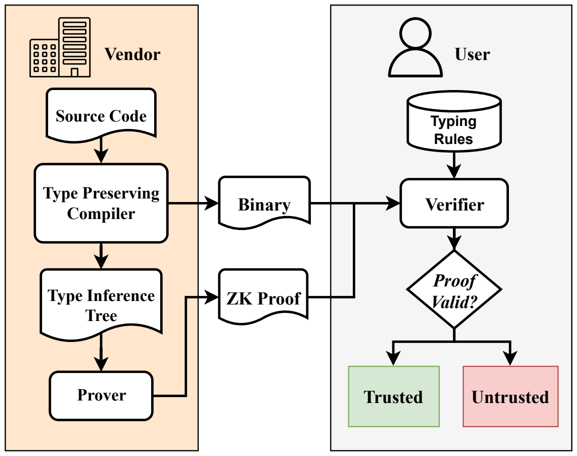

Fig. 1. Verification process of a binary with zkTAL

To generate the proof, the vendor first compiles their source code using a type-preserving compiler, resulting in assembly code annotated with types for the target architecture. These type annotations are then used to construct a type inference tree. If this tree is valid, it demonstrates that the assembly code is correctly typed and, therefore, memory-safe according to the guarantees of the type system. This type inference tree is then encoded within zero-knowledge, and is given to a zkSNARK prover, which then generates a cryptographic proof that witnesses the correct type derivation without revealing any information about the proof. Finally, the vendor ships the assembled binary, stripped of the type annotations, along with the corresponding zkSNARK proof.

On the user's side, the end-user receives the binary, the cryptographic proof, and a public specification of the typing rules. These components serve as input to the zkSNARK verifier, which then accepts or rejects the proof. If accepted, the binary can be considered trustworthy, and the user gets a cryptographic guarantee that the vendor successfully proved the delivered binary correct.

### A. Zero-Knowledge Encoding

A zkSNARK [3], [4] is a cryptographic proof system that allows a prover to convince a verifier of the validity of a statement without revealing any additional information beyond the fact that the statement is true. In zkTAL, the prover seeks to demonstrate to the verifier that they possess a valid proof of the binary's correctness, namely, a type derivation in the form of a type inference tree.

To encode typing checking within a zkSNARK, we must first encode it into an arithmetic circuit – a collection of addition and multiplication gates where values range over a finite field. Such an encoding is quite restrictive – most importantly, it means we cannot use unbounded loops or recursion to write our type-checker. The arithmetic circuit takes as input an encoding of the type-derivation tree that proves correctness of the binary. To supply this tree, we have to encode it in terms of a bounded number of field elements or arrays thereof. For this, we follow the approach of zkPi [5], a zkSNARK for Lean Theorems, and encode the type inference tree as a linear array of sub-derivations, linked by pointers. To check the correctness of these derivations, we have to retrace the type derivation of the original tree in the linearized array. To check a given derivation, we first check the derivation individually, and then recursively check all sub-derivation it depends on. We encode this dependence via pointers that point to the array location of each sub-derivation. For each typing judgment, we also include a reference to the typing rule it used. We provide typing rules as public input to the verifier who can therefore check that only valid typing rules where used in the proof. Since we cannot use unbounded loops or recursion in circuits, we have to unroll all loops in the recursive traversal of the linearized array.

### B. Typed Assembly Language

By targeting assembly language directly for type checking, rather than source code, we extend type safety guarantees directly to the binary level. These guarantees depend on the underlying type system; for example, Rust [2] demonstrates that a type system can enforce memory safety at the source code level. At the assembly language level, previous work by Morrisett et al. [6] has shown that a type-safe assembly language can be accompanied by a formal soundness proof, establishing that successfully type-checked programs have both memory and control flow safety.

Inspired by TALx86 [7] an implementation of a type system for x86 assembly, we decided to adapt the type system presented for TAL-1 [8], and use it as the basis for a future RISC-V type system. TAL-1 is a typed assembly language designed to enforce memory safety and control-flow integrity at the assembly level. Similar to Rust, it achieves these guarantees through a type system that manages memory operations using two different pointer types: *shared data pointers*, which maintain type invariance to safely handle aliased data, and *unique data pointers*, which ensure exclusive ownership, allowing controlled modifications to memory objects. TAL-1 also uses polymorphic types to effectively handle aliasing concerns, improving overall memory safety. The typing rules then ensure that all memory accesses and control-flow transitions strictly adhere to type constraints, eliminating many classes of unsafe behavior typically associated with low-level code.

### C. Challenges

Implementing zkTAL involves addressing several significant challenges. First, adapting and extending an existing type system to the RISC-V instruction set is itself a substantial undertaking. Combining this effort with the requirement to encode the verification of the type inference tree in a zero-knowledge proof system further increases complexity.

Regarding the type system, the main challenge is adapting the relatively simple type system of TAL-1 to the RISC-V instruction set. This adaptation involves redesigning heap memory and pointer types to more accurately reflect real hardware, defining typing rules for the new RISC-V instructions, and introducing additional types to handle varying data sizes.

On the zkSNARK side, the primary challenge is developing efficient encodings of the type-inference tree and typing rules, in order to ensure the verification circuits are compact which makes verification fast.

## III. Preliminary results

We have developed a preliminary proof-of-concept implementation of zkTAL using Noir,[1] a zkSNARK framework. Specifically, we implemented TAL-0 [8], a simple typed assembly language with control flow safety in Rust. From this implementation, we generated type-inference trees for typed assembly programs, and encoded these trees as arrays consisting of hashed terms and pointers. Using these encodings as a witness, we constructed a custom verification circuit in Noir, designed to validate the correctness of the encoded type inference trees. We then generated zkSNARK proofs using the Barretenberg backend, and successfully verified in zero-knowledge that the programs were correctly type-checked.

[1] https://noir-lang.org/

## References

[1] M. Cadariu, E. Bouwers, J. Visser, and A. Van Deursen, “Tracking known security vulnerabilities in proprietary software systems,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, (Montreal, QC, Canada), pp. 516–519, IEEE, Mar. 2015.

[2] N. D. Matsakis and F. S. Klock, “The rust language,” in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, (Portland Oregon USA), pp. 103–104, ACM, Oct. 2014.

[3] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer, “The Hunting of the SNARK,”

[4] J. Liang, D. Hu, P. Wu, Y. Yang, Q. Shen, and Z. Wu, “SoK: Understanding zk-SNARKs: The Gap Between Research and Practice,”

[5] E. Laufer, A. Ozdemir, and D. Boneh, “zkPi: Proving Lean Theorems in Zero-Knowledge,” 2017.

[6] G. Morrisett, D. Walker, and K. Crary, “From System F to Typed Assembly Language,”

[7] G. M. K. Crary, N. Glew, D. Grossman, and R. Samuels, “TALx86: A Realistic Typed Assembly Language,”

[8] B. C. Pierce, ed., *Advanced topics in types and programming languages*. Cambridge, Mass: MIT Press, 2005.