MTZK: Testing and Exploring Bugs in Zero-Knowledge (ZK) Compilers

Dongwei Xiao, Zhibo Liu*, Yiteng Peng and Shuai Wang*
The Hong Kong University of Science and Technology
{dxiaoad, zliudc, ypengbp, shuaiw}@cse.ust.hk

***Abstract*—Zero-knowledge (ZK) proofs have been increasingly popular in privacy-preserving applications and blockchain systems. To facilitate handy and efficient ZK proof generation for normal users, the industry has designed domain-specific languages (DSLs) and ZK compilers. Given a program in ZK DSL, a ZK compiler compiles it into a circuit, which is then passed to the prover and verifier for ZK checking. However, the correctness of ZK compilers is not well studied, and recent works have shown that de facto ZK compilers are buggy, which can allow malicious users to generate invalid proofs that are accepted by the verifier, causing security breaches and financial losses in cryptocurrency.**

In this paper, we propose MTZK, a metamorphic testing framework to test ZK compilers and uncover incorrect compilations. Our approach leverages deliberately designed metamorphic relations (MRs) to mutate ZK compiler inputs. This way, ZK compilers can be automatically tested for compilation correctness using inputs and mutated variants without requiring manual intervention. We propose a set of design considerations and optimizations to deliver an efficient and effective testing framework. In the evaluation of four industrial ZK compilers, we successfully uncovered 21 bugs, out of which the developers have promptly patched 15. We also show possible exploitations of the uncovered bugs to demonstrate their severe security implications.

I. Introduction

Zero Knowledge (ZK) cryptographic proof systems [48] have a wide range of applications in various fields. From the *privacy* perspective, Zero Knowledge Proof (ZKP) systems are used to provide confidential computation and data sharing in the financial and healthcare sectors [49], [45], [80], [5], where they can facilitate verifying transactions without revealing sensitive information. Moreover, a promising trend is to use ZKP systems to achieve *scalability* in blockchain systems [37], [22], where they allow users to create private transactions and scale the blockchain with protocols like zkRollups [23].

Given the general difficulty of writing ZK programs in circuit representation, the community has developed domain-specific languages (DSLs) and ZK compilers to automatically translate a program P written in DSLs into circuit R. Nevertheless, there is no decision procedure for proving $P \Leftrightarrow R$ for an arbitrary computation P [83]. That is, existing ZK compilers are facing an intractably hard problem and are prone to bugs. Moreover, de facto ZK compilers often comprise dozens to hundreds of thousands of lines of code [105], [19], covering a complex compilation and optimization pipeline, including the compiler frontend, backend, and circuit generation component. All these sophisticated components make ZK compilers a highly complex system that demands careful design consideration. Notably, since ZK compilers have been applied in various sensitive scenarios, from financial sectors to blockchain systems, the correctness of ZK compilers is of paramount importance. Bugs in ZK compilers, in turn, can lead to privacy leakage [62] and financial losses [8], [97], [55] of several million dollars worth of cryptocurrency.

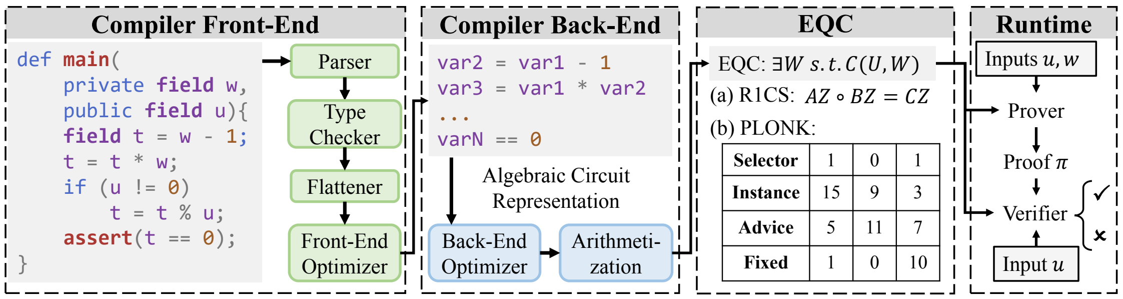
This work presents MTZK, the first automated, systematic metamorphic testing (MT) framework for ZK compilers. MTZK tackles black-box scenarios, allowing testing of commercial, off-the-shelf ZK compilers and holistically uncovering bugs in the full pipeline of ZK compilers. Instead of capturing obvious "crash" behaviors (as conventional software fuzzing does), MTZK uncovers incorrect compilation outputs (logic bugs) residing in the complex compilation pipeline of ZK compilers. This is achieved by performing MT, an invariant property-based testing method that relies on mutation rules referred to as metamorphic relations (MRs). MT alleviates the difficulty of determining the expected outputs of test inputs (which often requires human annotation and is often prohibitively expensive) by verifying the target software's *behavior consistency* under MR-mutated test inputs. As a result, ZK compiler testing becomes substantially more flexible without requiring manual intervention.

We design two novel MRs to transform ZK programs and check ZK compiler output consistency with respect to the original and mutated ZK programs. Our MRs are designed as semantics-preserving, meaning a seed ZK program and its mutated version should have identical output. These two MRs are, however, tailored to mutate certain ZK program features that impose a major "stress" on the compilation and optimization pipeline of ZK compilers and, thus, are more likely to expose bugs. By applying our semantics-preserving MRs on ZK programs, we can expose ZK compiler bugs by automatically checking the output consistency of circuits compiled from ZK programs and their mutated versions.

We implement MTZK targeting four state-of-the-art ZK compilers: ZoKrates [105], Noir [64], Cairo [21], and Leo [4]. MTZK generates 30K ZK programs in total to test ZK compilers. During our testing campaign, we detected 21 bugs in total, 15 of which have been responsibly fixed by the developers. Moreover, we also make the first attempt to launch hazard analysis towards the uncovered ZK compiler bugs and show

*Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230530
www.ndss-symposium.org



```
def main(
    private field w,
    public field u){
    field t = w - 1;
    t = t * w;
    if (u != 0)
        t = t % u;
    assert(t == 0);
}
```

```
var2 = var1 - 1
var3 = var1 * var2
...
varN == 0
```

Selector	1	0	1
Instance	15	9	3
Advice	5	11	7
Fixed	1	0	10

Fig. 1: High-level workflow of the ZK compilers and the proof systems.

that many of our bugs can presumably induce severe security breaches, e.g., backdoors can be stealthily inserted by the ZK compiler in the compiled ZK applications. In sum, we make the following contributions:

- This work introduces a new focus to launch systematic testing for ZK compilers. We design testing methods to expose logic bugs in ZK compilers. The detected bugs are highly critical and can largely change the behavior of compiled ZK circuits, leading to security breaches.
- Our framework, MTZK, performs MT to test ZK compilers. Two novel MRs are designed to mutate ZK programs and stress ZK compilers. We incorporate various design principles and optimizations to deliver effective testing.
- We tested four industry-leading ZK compilers and found 21 bugs, 15 of which have been promptly fixed. We also analyze the security implications of the uncovered bugs and find that many can lead to severe security breaches. As proof of concept, we demonstrate examples of exploitation of the uncovered bugs.

Artifact Availability. We release and will maintain the codebase of MTZK at [110] to boost future research.

II. Preliminary

ZK Systems. In general, the goal of a ZK system is to allow users to prove statements while using but not revealing some secret information. Such a process generally contains two parties: a *prover* and a *verifier*. The goal of the prover is to generate proof that he or she knows a secret witness w satisfying a constraint $C(u, w)$ for some public parameters u that are known by both the prover and the verifier. The verifier will then verify the proof (with high probability) if and only if the prover actually knows the secret witness w satisfying the constraint. We further say the proof system is *zero knowledge* if the verifier cannot learn anything about w other than the fact that $C(u, w)$ is satisfied.

Take a bank system as an example: a bank enforces that any customer must own over 1,000$ in order to open a credit card. The prover is the customer, and the verifier is the bank. The customer proving process aims to prove the relation $C(u, w) : w \geq u$ without revealing the customer's assets. Throughout the paper, we will use the symbol w to denote the private inputs of the prover, and u to denote the public inputs.

The "Runtime" column of Fig. 1 depicts a typical workflow of ZK systems. First, the prover and the verifier take as input the encoding of a constraint C that they want to prove, and engage in a ceremony to set up common parameters (omitted in Fig. 1). Next, based on the common parameters and public inputs u, the prover generates a proof π as a testimonial to private inputs w. The verifier will take the public input u and verify the proof π. The verification passes if and only if π indicates that w satisfies the constraint $C(u, w)$. To ease presentation and use common notations, we state that the $C(u, w) = \text{SAT}$ (or UNSAT) if the constraint is satisfied (or not satisfied) under public inputs u and private inputs w, and accordingly, the verification *passes* (or *fails*).

Existentially Quantified Circuits (EQC). In general, constraint C is represented in a special format of arithmetic circuits named EQC. EQCs are circuit-like, meaning they comprise connections between wires and signals and cannot contain loops or branching. In addition, EQCs are *existentially quantified*: they can contain variables whose value depends on the other variables in the circuit to make the overall output of the circuit true. Take a boolean circuit with the form $\exists A\ s.t.\ A \vee B$ as an example. In this case, the boolean variable A is existentially quantified. When B is false, A must be true to make the constraint satisfied. In the context of ZK systems, the constraint C is typically expressed as follows:

$$\exists w\ s.t.\ C(u, w) = \text{SAT} \tag{1}$$

The secret w in Eq. (1) is existentially quantified. EQC is particularly suitable for ZK systems since the prover should not directly reveal the secret w to the verifier but rather prove that there exists a w satisfying the constraint C.[1]

ZK Protocols. ZK protocols are the core of ZKP systems. They are the algorithms that allow the prover to generate a proof π and specify the details of the proving and verification stage. To date, major ZK protocols include SNARKs [51], STARKs [11], and Halo2 [15]. Different ZK protocols feature different proof sizes and verification times.

ZK protocols are typically designed to work with EQCs. Different ZK protocols may have different requirements for EQCs' formats. For instance, SNARKs require the EQCs to be in the form of *rank-1 constraints* (R1CS) [51], while STARKs accept *Algebraic Intermediate Representation* (AIR) [11], and Halo2 [15] requires *PLONK* [44] format. Fig. 1 shows the EQC format of the two protocols, in which R1CS represents EQCs as a matrix multiplication form of $AZ \circ BZ = CZ$ while PLONK represents EQCs as a table.

[1]For simplicity, we assume "the prover can prove that there exists w" is equivalent to "the prover knows such w" in this paper.

ZK DSLs and Compilers. Directly writing EQCs in the format of R1CS or PLONK is tedious and error-prone. To ease the burden of writing EQCs, the community has developed DSLs like Zokrates [34], Cairo [47], Noir [64], and Leo [28], to allow users to write their computation in a DSL program. Given a DSL program P that takes public inputs u and private inputs w and outputs y, the ZK compiler compiles P into an EQC $C(u, w, y)$, such that the following property holds:

$$P(u, w) = y \iff \exists w \ s.t. \ C(u, w, y) = \text{SAT} \quad (2)$$

Here, we slightly abuse the notion of C to take three inputs instead of two. Moreover, to ease presentation, this paper refers to EQCs as "executables," and the process of proving and verifying the satisfiability of EQCs as "execution."

ZK DSL Fig. 1 presents a DSL program from the Zokrates language, which is heavily inspired by Rust. The source program in the example takes as input a public input variable `u` and a private input variable `w`. The type of input variables is `field`, a type representing finite field numbers and being widely used in ZK systems. Public variables are visible to both the prover and the verifier, while the private variables are provided by the prover and are not visible to the verifier. Instead of writing circuit-like constraints directly, developers of ZK applications can declare the constraints that should be satisfied to yield an SAT result in verification in the high-level DSL with the `assert` statements. The constraints in the DSL would later be compiled into constraints in C in the EQC format. We denote the set of constraints in the high-level program P as $\phi_1(u, w), \phi_2(u, w), \ldots$. For instance, the sample program in Fig. 1 computes that $w \times (w - 1)$ `mod` u (or $w \times (w - 1)$ when $u = 0$), and asserts the result to be zero.

The program output y is made up of the return value from the `return` statement and, more importantly, the boolean values of all the `assert` statements indicating whether the constraints are satisfied. In the sample program of Fig. 1, the output y consists of merely the boolean value of the `assert(t == 0)` statement, as the program's return type is `void`.

ZK Compiler Frontend. As depicted in Fig. 1, compiling a ZK program P involves a complex compilation and optimization pipeline. P is first parsed into an abstract syntax tree (AST). ZK compilers perform type checking and semantic analysis on the AST to ensure the program is well-formed. Notably, one core task of type checking is to mark the *visibility* of variables, i.e., whether a variable is *public* or *private*. Typically, information-flow (taint) analysis is performed to identify all variables directly or indirectly derived from the private inputs [91]. The AST would then be transformed into a frontend intermediate representation (IR) that is suitable for frontend analysis and optimization. The IR is then sent to a "flattener" to convert into the circuit format. Because circuits are stateless, the flattener first transforms the IR into the static single assignment (SSA) form, unrolls loops, inlines all functions, and also transforms if-statements into conditional selectors. The output from the flattener is further optimized by the frontend optimizer, which performs optimizations like common subexpression elimination, constant propagation, and bit reorganization.

ZK Compiler Backend. After the frontend stage, the program is now in an *Algebraic Circuit Representation*, as shown in the second column of Fig. 1. Such representation can declare circuit signals with = and specify the constraints between them with ==. The backend optimizer then optimizes the circuit representation. The constraint checking under ZK is highly costly and is at least three orders of magnitude slower than directly checking the constraints without ZK [107]. Hence, it is crucial for the optimizer to reduce the number of constraints. Various optimization strategies are implemented at this step. For instance, the linearity reduction optimization [81] reduces the number of constraint variables in the R1CS system; the liveness analysis [47], [19] would remove unused constraints. The optimized circuit finally goes through an arithmetization process to lower integers and arrays to finite field numbers that EQCs can accept.

III. Motivation

Challenges in Developing Correct ZK Compilers. The computation theory has proved that:

There is no decision procedure for proving $C(u, w, y) \Leftrightarrow P(u, w) = y$ for an arbitrary computation P [83].[2]

In complexity-theoretic terms, the circuit $C(u, w, y)$ is *non-deterministic*, while the program in the high-level language $P(u, w) = y$ is *deterministic* [6]. Intuitively, given a concrete circuit C and the value of u, y, the value of w can only be "guessed," e.g., by brute-forcing all possible w to find a concrete value of w that satisfies the constraints enforced by C; in contrast, given the high-level program P and the inputs u, w, the value of y can be readily obtained by executing the program and obtaining its execution output.

According to the theorem, ZK compilation faces an intractably hard problem, and therefore, it is challenging to verify if compiled circuits conform to the semantics in high-level ZK programs. Thus, developing *correct* ZK compilers is difficult, and developers have limited tools to check the correctness of compiled executables. To the best of our knowledge, systematic and automated approaches to testing ZK compilers are lacking.

Vulnerability & Significance. Given the industrial adoption of ZK protocols in critical scenarios (e.g., Ethereum), bugs related to ZK proofs can lead to severe consequences, such as catastrophic monetary loss. A bug [55] in ZCash [117] results in generating infinite ZCash coins. Also, a bug in implementing constraints [8] in Aztec [9], a popular L2-chain network on Ethereum, could allow attackers to spend a banknote multiple times. The severity of those bugs demonstrates the significance and urgency of designing a specific method to detect ZK-specific bugs.

ZK compilers are a critical component in ZK systems. High-level source programs of ZK applications are compiled into circuits by ZK compilers. Bugs in ZK compilers can thus lead to a well-formed high-level program being compiled into an incorrect circuit, which can result in security vulnerabilities in the ZK system. Compiler bugs can lead to backdoors [106] or exploits [53], [54]. As such, it is crucial to ensure the correctness of ZK compilers.

[2]Although it is possible to prove the equivalence for a specific computation in an ad-hoc manner, e.g., enumerating all possible u, w, y, for proving **all** computations, the problem is undecidable.

Motivating Example. We examplify how ZK compiler bugs can lead to incorrectly accepted proofs with a real bug found by MTZK in Fig. 2. In the given example, the program takes two inputs a and b, both of which are of type `field`, which is special data type representing a finite field element. The variable a is private and is provided by the prover, while b is public and is pre-agreed upon by both parties. The program has a constraint enforcing that the value of a should be less than that of b. During runtime execution of the program, the prover proves to the verifier that the input value of a is less than that of b. The program is then compiled into a ZK circuit by the ZK compiler. Despite the source code program being well-formed, the ZK compiler incorrectly compiles the program, leaving an exploitable backdoor in the compiled circuit. A malicious prover can fake a proof that a is less than b, while in reality, a is greater than b. The verifier, however, accepts the invalid proof, leading to a security breach. The root cause of the backdoor in the compiled circuit is not in the source code but in the compiler itself. As such, it is crucial to develop systematic testing approaches to detect bugs in ZK compilers.

```
fn constraint_lt(a: private field, b: public field) {
    assert(a < b);
}
```

Fig. 2: A motivating example of a ZK compiler bug.

Contributions. Given the importance of ZK compilers and the lack of systematic testing approaches, we propose MTZK, the *first* systematic testing framework for ZK compilers. To comprehensively stress-test ZK compilers, MTZK features MRs that focus on ZK-specific features, including the constraint and private data aspects, as mutating them can effectively exert pressure on many ZK-specific optimizations and translations, as well as the standard optimization routines. We test four widely used industrial ZK compilers (see Section VI), and we illustrate potential exploitations to demonstrate the severity of the uncovered bugs. We believe that our work can help developers improve the robustness of ZK compilers and enhance the security of ZK systems.

IV. Overview of MTZK

A. Study Scope

The proposed MTZK tests the correctness of ZK compilers. In general, MTZK is able to uncover two types of bugs (although its main focus is on the second type):

1) *Compilation failures.* This type of error has noticeable symptoms, such as throwing exceptions or causing crashes. These failures may lead to memory-based exploitations or denial-of-service (DoS) when the compiler is accessible by attackers. They are, however, easier to detect (e.g., using fuzzing) because of their obviously erroneous behaviors.
2) *Logic errors.* This type of bug does not directly lead to crashes or exceptions. Instead, it causes the compiler to generate incorrect executables. Such errors can be very dangerous since they can cause the verifier to accept invalid proofs and thus induce security breaches. Overall, while we also detect compilation failures, our focus is on logic errors due to their stealthiness and higher severity.

B. Testing Oracle Design

Challenges. A key challenge in detecting logic errors is the lack of ground truth (i.e., *testing oracle*). According to the theorem mentioned in Section III, it is generally impossible to determine whether a generated EQC C from the ZK compiler is equivalent to the high-level program P for every possible P (except relying on extensive manual inspection, which is prohibitively expensive). In this work, we adopt *metamorphic testing* (MT) [26] as an automated method to detect logic errors in ZK compilers. MT is a testing technique that alleviates the testing oracle issue by checking properties that must hold for the correct implementation of the program. For instance, an algorithm P that finds the shortest path in an undirected graph with only positive edge weights must satisfy a property that the shortest path from A to B discovered by P should be the same as the shortest path from B to A. We can thus test P by checking if it always satisfies the property with randomly decided inputs A and B; violations of the property indicate bugs in P. In short, this property and its derived testing oracle are referred to as *metamorphic relation* (MR). In this paper, we identify two novel MRs in the context of ZK programs and use them to detect logic errors in ZK compilers.

MR Design. We design two MRs to detect logic errors in ZK compilers. The primary purpose of ZK systems is to prove the satisfiability of constraints over program inputs without leaking the private inputs. Thus, our MRs are particularly designed to mutate the constraints in ZK programs and visibility (i.e., a variable is public or private) of ZK program variables, with the expectation that the MRs can effectively stress the compiler to uncover more severe bugs. Technically speaking, an MR is composed of a mutation and its associated invariant property. We now introduce the two MRs in detail.

MR_{SIM}: Satisfiability-Invariant Mutation (SIM) This MR inserts a set of always-satisfied constraints w.r.t. inputs into the source program and asserts that the execution of the compiled program should also yield SAT. Given the constraint $\phi_1, \phi_2, \ldots, \phi_n$ in the source program P and its corresponding compiled constraint C in the executable, when all the ϕ yield SAT under inputs u, w, C should also result in SAT. Formally:

$$\bigwedge_{i=1}^{n} \phi_i(u, w) = \text{SAT} \implies C(u, w) = \text{SAT} \qquad (3)$$

The format of ϕ and C are highly different (`assert` statements in Rust-like ZK programs vs. EQC), and ZK compilers usually perform a series of non-trivial transformations to compile ϕ to C. Also, since the number of constraints in EQC significantly affects the proving and verification speed, ZK compilers employ a series of optimization passes to reduce the number of constraints. Given that said, MR_{SIM} faces obstacles in its implementation (since we cannot automatically evaluate the satisfiability of constraints in ZK high-level programs); see our solution in Section V.

MR_{IVM}: Information Visibility Mutation (IVM). This MR mutates the visibility of program variables and asserts that the program's functionality should not be affected. Given the public inputs u and private inputs w, when exchanging the input visibility, i.e., specifying u as private and w as public, the semantics of the program should be unchanged. MR_{IVM} mutates the visibility of the inputs and asserts that

the functionality of the program should not be affected. As introduced in Section II, ZK compilers implement a set of compilation and optimization passes to perform information flow (taint) analysis and identify variables that are "tainted" by private inputs; those tainted ones are also marked as private data. Moreover, ZK compilers enforce different compilation/optimization schemes for public and private data, and computations involving private data are typically much more complex to optimize than those involving public data. This way, by mutating the input visibility, we anticipate effectively stressing the analysis routines of the compiler and their subsequently applied compilation/optimization passes.

C. Usage Scenario and Clarification

Main Audiences. The main audiences of this work are ZK compiler developers and vendors. Our work helps them to test their compilers before release. As shown in Section VII, developers of ZK compilers responsibly responded to our bug reports and promptly fixed the bugs, indicating the utility of our work. MTZK uncovers subtle logic errors in ZK compilers, which are hard to find with existing fuzzing testing techniques. The findings are highly critical and often indicate potential attack vectors that malicious users can exploit in compiled ZK executables [55], [7]. MTZK is, however, not designed to be used by malicious users to exploit ZK compilers. We provide further clarification on the ethical concerns in Appendix B.

Compiler Bugs vs. Application Bugs. We are aware of recent advances in detecting bugs in ZK applications [108], [75]. However, they neglect the bridge (i.e., ZK compilers) between high-level ZK programs and their compiled circuits. Even if the high-level ZK programs are correct, the compiled circuits may still be incorrect due to stealthy bugs in ZK compilers. Our work opens a new frontier in testing ZK compilers, which are highly complex and hard to verify. Application bugs stem from errors or flaws in the source code written by developers, such as misuse of certain components or misspecification of constraints. Compiler bugs, on the other hand, arise from errors in the compiler software itself. Due to the distinct nature of the two types of bugs, they require different approaches to detect. While application bugs primarily affect the functionality of a single program, compiler bugs have the potential to introduce errors in any program compiled using the faulty compiler, making their impact more widespread and challenging to isolate.

Another recent work by Pailoor et al. [83] captures under-constrained R1CS circuits. Although an under-constrained circuit can be due to both application and compiler bugs, all the eight bugs they found are application bugs. In contrast, by designing a testing pipeline tailored for ZK compilers, our approach effectively detects 21 compiler bugs. Moreover, Detection techniques for under-constrained R1CS circuits cannot be directly applied to other protocols like PLONK [83], [95]. MTZK is agnostic to the underlying ZK protocols since it is a black-box testing tool, and our four testing targets cover three different ZK protocols (see Section VI). In addition, MTZK is able to capture a variety of logic bugs other than underconstraints, as shown in Section VII. We provide a further comparison of our bugs with those found by ZK application bug hunters in Appendix A.

Fuzzing vs. Metamorphic Testing. Fuzzing is widely adopted in software testing [115]. However, fuzzing relies on obvious symptoms (e.g., crashes or memory access violations) to detect bugs. Without explicit oracle, fuzzing cannot detect logic errors, which can be more subtle and dangerous. In contrast, metamorphic testing (MT) [25] is equipped with carefully designed MRs as testing oracles and has helped to reveal numerous logic bugs in production compilers like GCC [67], [68], [98]. MTZK thus employs MT to detect logic errors in ZK compilers.

EMI vs. ZK-Specific MRs. Equivalence Modulo Inputs (EMI) [67], [68], [98] is an impactful MT technique that performs semantics-preserving transformations towards seed programs and checks equivalence of outputs from the mutants. Despite its success in finding bugs in production compilers like GCC and LLVM, we clarify that EMI is not effective at detecting bugs in ZK compilers due to the unique features of ZK programs. The dead code insertion strategy, i.e., a representative mutation of EMI, typically requires pinpointing uncovered code components in a program and inserting code that should not be executed. However, ZK programs require special syntax/semantics and will always execute both branches (no "uncovered code"). Moreover, ZK programs focus on proving constraints, while EMI mutates program control/data flow without introducing any constraints. Note that a ZK program without any constraint will trivially pass verification and thus cannot help uncover any compiler logic bug. Overall, based on ZK's unique computation paradigm, we introduce two novel MRs tailored for ZK compilers in Section IV-B.

White-Box vs. Black-Box. MTZK considers a black-box testing scenario where we mutate ZK compiler inputs and check compiler output consistency (using our MRs in Section IV-B) to decide whether the compiler is buggy. We clarify that "black-box" does not necessarily mean that the source code of the compiler is not available, but instead, we do not rely on the compiler's internal implementation details. ZK compilers vary significantly in terms of the ZK protocols they support, the programming languages in which they are developed, and the compilation and optimization routines they employ. The black-box view enables us to test any ZK compiler without huge engineering efforts to adapt our tool to the compiler's internal structure. Such a setting is also aligned with many existing testing tools for software compilers [114], [98]. We leave the exploration of white-box testing (e.g., relying on certain coverage criteria in the compiler codebase) for future work.

Testing vs. Verification. To the best of our knowledge, MTZK is the first *testing* approach toward ZK compilers, addressing a timely demand of our community and industry. MTZK shares a similar testing-based approach with quality assurance tools for critical systems, such as CPUs [56], [46], databases [89], [59], [60], [90], and operating systems [65], [16], to detect errors instead of proving their absence. MTZK cannot offer a formal guarantee: when MTZK has no findings, we cannot conclude that a ZK compiler is free from bugs. Nevertheless, the testing approach delivered by MTZK is precise and speedy, with almost no false positives. More importantly, as a testing tool, MTZK provides defect-triggering inputs, which enable developers to debug and fix the uncovered bugs (see Section VII for details). In contrast, we notice emerging research that aims to formally verify the correctness of ZK compilers [41], [82], [28] (details in Section IX). As a common and textbooklevel concept, testing and verification are two complementary approaches to software quality assurance. The former is more practical and scalable, while the latter is more rigorous (but may have false positives). Overall, our work meets the high demand for testing ZK compilers, especially given that modern ZK compilers can easily have over 100K lines of code with complex compilation and optimization routines. These features make the formal verification of ZK compilers extremely challenging.

V. Design of MTZK

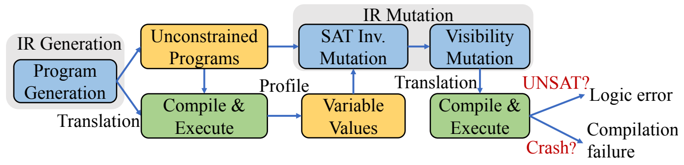


Fig. 3: The testing pipeline of MTZK.

Design Overview. Fig. [3] shows the testing pipeline of MTZK, which consists of three testing stages (shown in blue boxes):

① *Random Program Generation.* This phase randomly generates source programs of ZK DSLs. To enhance generality and ease the implementation, we first perform the generation using our customized IR, and then concretize the IR code into the specific syntax of a target ZK DSL. The details of our designed IR are in Section [V-A]. The randomly generated programs do not contain any constraint statements.

② *Testing with MR_{SIM}.* This phase uses MR_{SIM}; it mutates programs generated in ① and inserts constraints that are determined to yield an SAT result. To do so, we first compile and execute the program from ①, and profile the variable values. We then propose a set of rules (given in Section [V-C]) to generate such constraints based on the profiled values.

③ *Testing with MR_{IVM}.* This testing stage employs MR_{IVM}; it mutates the information visibility of variables in the mutated ZK program from MR_{SIM} (i.e., the output programs of ②). Following the definition of MR_{IVM}, we mutate the information visibility of the program input variables and assert that the mutated programs retain the original constraint-checking results (i.e., SAT). If the proof given by the prover is rejected by the verifier (i.e., "UNSAT") in the verification phase, we deem it a bug in the ZK compiler.

We will send the resulting programs from each of the three phases to the tested ZK compiler for compilation and execution. If a crash or exception is detected, we deem it a compilation failure; rather, if the compiler outputs violate either MR_{SIM} or MR_{IVM}, we deem it a logic error.

A. IR Design

This paper first defines a general IR λ_{gen} that subsumes mainstream ZK DSL features. We then define the program generation strategy over the IR, and finally, we concretize each IR program into the target DSL. The syntax of the λ_{gen} is demonstrated in Fig. [4]. It supports basic data types φ, e.g., `field`, that frequently appear in ZK programs. Operations on the data types are also supported, including arithmetic

Program	P	$::=$	$s; P \mid s;$
Statement	s	$::=$	`if` e `then` $\{s^*\}$ `else` $\{s^*\}$ $\mid$ `for` x `in` $n..n$ $\{s^*\}$ $\mid$ vd $x = e$ $\mid$ `assert` e
VarDef	vd	$::=$	`let` $x : \tau = e$
Variable	x	$::=$	*valid variable name*
BinaryOp	$\otimes$	$::=$	$+ \mid - \mid \times \mid$ `div` $\mid$ `mod` $\mid$ $< \mid > \mid \leq \mid \geq \mid == \mid \neq$ $\& \mid \mid \mid \oplus \mid \wedge \mid \vee$
UnaryOp	$\diamond$	$::=$	$\neg \mid - \mid$ `bit_not`
BasicType	φ	$::=$	`int` $\mid$ `uint` $\mid$ `bool` $\mid$ `field`
PrivacyType	ϵ	$::=$	`public` $\mid$ `private`
Type	τ	$::=$	$\{\nu : \varphi \mid \nu : \epsilon\}$
Value	v	$::=$	`true` $\mid$ `false` $\mid$ n
Number	n	$::=$	$0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots$
Expression	e	$::=$	$v \mid x \mid e_1 \otimes e_2 \mid \diamond e$

Fig. 4: Syntax of the λ_{gen}.

operations (e.g., addition $+$), bit-wise operations (e.g., exclusive-or $\oplus$), as well as logic operations. The IR subsumes most of the commonly seen statements, including variable declaration (`let` $x : \tau = e$), and control flow statements (e.g., `if` and `for`).

ZK DSLs have a clear distinction over the information visibility of the variables: ZK protocols enforce that all private variables, including explicitly declared ones and intermediate variables, are only visible to the prover and should be protected with cryptographic primitives during the ZK protocol execution. Contrarily, public variables are visible to both the prover and the verifier. To accommodate the distinction between different information visibility levels, the type system of λ_{gen} is designed as a *refinement type system* [109], [42]. A refinement type τ has the form $\{\nu : \varphi \mid \nu : \epsilon\}$, where ν is a symbolic placeholder for the variable name, φ is the basic type of the variable, and ϵ encodes the visibility of the variable. The privacy type ϵ can be either `public` or `private`, indicating whether the variable is public or private. For instance, the type $\{\nu :$ `int` $\mid \nu :$ `private`$\}$ indicates that the variable is an integer and is private. The refinement type notations in λ_{gen} facilitate designing an information visibility mutation scheme (as in MR_{IVM} discussed in Section [V-D]) to effectively change the visibility of the variables in ZK programs.

B. Program Generation

We generate IR code following the IR λ_{gen} and then implement a translator to concretize each generated IR code into a ZK program based on the grammar of the target ZK DSL. Program generation produces programs by recursively applying production rules of the syntax grammar starting from a start symbol. In our case, the syntax grammar is λ_{gen}, and the start symbol is P. We now describe the internals of our program generator used to produce programs in λ_{gen}.

Local Context Compatability. To ensure a variable is defined before use, our program generator maintains a *type context* Γ, which stores a mapping from the variable name x to its type τ:

$$\Gamma ::= \{x_1 : \tau_1, x_2 : \tau_2, \ldots\} \quad (4)$$

When the generator refers to a variable name, it consults the context to determine the variable type under the current scope. We use the notion $\Gamma(x) = \tau$ to denote that the variable x has type τ in the context Γ. We must ensure that any reference to a variable lies within its scope. Denote the type context at line l as Γ_l, and the domain of the types of a context at line l as $dom\ \Gamma_l$:

$$dom\ \Gamma_l = \{v \mid v : \tau \in \Gamma_l\} \tag{5}$$

Given the line l that defines a variable x with type τ, the context Γ_l should contain the mapping $x : \tau$, i.e., $\Gamma_l(x) = \tau$. Furthermore, the contexts between lines l and $l + 1$ satisfy the following relationship:

$$\forall v \in (dom\ \Gamma_l \cap dom\ \Gamma_{l+1}) \setminus \{x\}, \Gamma_l(v) = \Gamma_{l+1}(v) \tag{6}$$

Variables should only be referenced if they are within the domain of the type context. Let $vars(e)$ be the set of variables referenced in an expression e. The expression e is semantically valid only if $vars(e) \subseteq dom\ \Gamma_l$.

Type-Guided Expression Generation. Semantically valid programs require valid types for operations. Our preliminary study showed that generating expressions blindly can result in many type-invalid statements that are rejected by the ZK compiler frontend, thus wasting testing effort. Hence, we propose a type-guided expression generation strategy to ensure the type validity of generated expressions.

To ensure the generated expressions are well-typed, our program generator adopts a *type-guided* generation approach to generate expressions. When generating a statement, the generator first randomly selects a type τ as the target expression type, then recursively produces well-typed candidate expressions compatible with the target type. MTZK uses a depth parameter to limit the maximum depth of the generated expression to avoid combinatorial explosion. The default maximum depth is 2, but can be configured by the user.

A simplified version of type-guided expression generation is shown in Algorithm 1. The procedure accepts the target type τ and the remaining expression depth d as arguments. If the remaining depth is 0 (a leaf node), it returns a random constant value with type τ. Otherwise, the SELECTOP procedure randomly selects an operator op conforming to the target type τ. The procedure NUMOPDS then determines the number of operands n_{opd} based on the operator op (unary, binary, etc.), and generates each operand opd_i recursively with its corresponding type τ_{opd}. Finally, it constructs an expression $expr$ that applies the operator op to all of its operands opd_i.

Algorithm 1 Type-Guided Expression Generation

```
 1: function GENEXPR($\tau, d$)
 2:     if $d = 0$ then
 3:         return GENCONST($\tau$)
 4:     $op \leftarrow$ SELECTOP($\tau$)
 5:     $n_{opd} \leftarrow$ NUMOPDS($op$)
 6:     for $i \leftarrow 1$ to $n_{opd}$ do
 7:         $\tau_{opd} \leftarrow$ OPDTYPE($op, \tau, i$)
 8:         $opd_i \leftarrow$ GENEXPR($\tau_{opd}, d - 1$)
 9:     $expr \leftarrow op(opd_1, \dots, opd_{n_{opd}})$
10:     return $expr$
```

Considerations for Program Validity. ZK DSLs have a set of requirements for a program to be valid. For instance, the division operator requires the divisor to be non-zero. We address those issues by ruling out invalid programs based on runtime exceptions. We notice that contemporary ZK compilers would throw exceptions hinting about certain subtle invalidities in an input program. Hence, we leverage the exception messages to rule out invalid programs. During testing, we perform a pre-filtering step to first compile the generated programs and check whether the ZK compiler throws any exception. When encountering an exception complaining about the input invalidity, we discard the input program and do not include it in the testing campaign. We list reasons for invalidities and the corresponding exception message patterns in Table XV in the appendix. After our testing campaign, we launched root cause analysis towards the uncovered error-triggering inputs, and we found that *none* of them was caused by the invalidity issue (see Section VII-B).

C. MR_{SIM}: Satisfiability-Invariant Mutation

Motivation. This section introduces the implementation of our first MR, MR_{SIM}. As encoded in Eq. 3, MR_{SIM} aims to mutate the randomly generated programs by inserting constraints. It stress-tests ZK compilers' ability to correctly compile a ZK program, including the constraints and all involved data and control flow, into correct constraints in low-level circuits. As mentioned in Section II, constraints encode the "primary functionality" of ZK programs, as ZK programs' outputs are typically derived from the constraints. Also, the number of constraints is a key factor that affects the performance of the ZK programs. Hence, correctly compiling and optimizing constraints are the key requirements of ZK compilers.

A constraint $\phi(u, w)$ in the source program specifies the relation that should be satisfied by the public inputs u and private inputs w. $\phi(u, w)$ is typically expressed in the form of `assert` e, where e is a boolean expression that (transitively) depends on the inputs. We will use the notion $\mathcal{C}(e)$ to represent a constraint statement in the source program.

Implementation Challenge. Holistically, MR_{SIM}, as denoted in Eq. 3, checks if constraint-checking results in the high-level ZK program are aligned with that of the compiled EQC; violating Eq. 3 denotes a bug. Despite its simplicity, MR_{SIM} is hard to implement directly, as it is challenging, if not impossible, to automatically obtain the constraint-solving results for arbitrary high-level ZK programs.

Thus, following the code generation phase in Section V-B, we propose to mutate each randomly generated program by generating and inserting constraints that are guaranteed to yield a "SAT" verdict, i.e., the secret inputs are guaranteed to satisfy the constraints. This way, we check if the execution of the compiled ZK programs also yields an "SAT" constraint-solving output. If not, we conclude that the ZK compiler has a bug. Benefiting from our deliberately designed constraint generation strategy (see below), we can automatically generate mutated test inputs and assert the correctness of ZK compilers.

Satisfiability-Invariant Generation. We first introduce necessary notions. Let the runtime environment that maps the variable names to their values be V. Such a mapping can be obtained by profiling the execution of the source program, e.g., by printing the values of variables after each variable assignment. Also, denote the evaluation of an expression e

under the runtime environment V as $V \vdash e \Downarrow v$. The notion of $V \vdash \mathcal{C}(e) \Downarrow v$ means that evaluating the checking $\mathcal{C}(e)$ under the runtime environment V successfully yields some value v, instead of causing a constraint failure exception.

We list MTZK's rules for generating constraints in Fig. 5. Overall, these rules facilitate generating the expression e in $\phi(e)$ that should lead to an SAT output during execution. C-EQ and C-NLT state that for a variable x whose value is v according to the runtime environment V, the expressions $x == v$ and $\neg(x < v)$ should be true. C-MAX leverages the type information and states that the value of a variable x should not exceed the maximum value of its type. C-TAUT is based on the tautology that the expression $x == x \times 1$ should always be true. C-AND and C-OR provide methods for expression composition. C-AND combines two true expressions e_1 and e_2, and constructs a new true expression $e_1 \wedge e_2$. C-OR is based on the intuition that in order for an OR expression to be true, only one of the two expressions needs to be true. Hence, we can combine a true expression e_1 and another randomly generated one e_2 to form a true OR expression $e_1 \vee e_2$. To clarify, the rules in Fig. 5 are not exhaustive; in our implementation, some rules (e.g., C-TAUT) can be initialized into several variants, with different operators or operands. We omit details for brevity.

Fig. 6 illustrates an example of generating constraints for a ZK program. Overall, it is challenging to obtain the satisfiability result for arbitrary constraints in a high-level ZK program, which may require symbolic execution and constraint-solving techniques [17]. For instance, the control/data flow in Fig. 6 makes it non-trivial to reason about constraint satisfiability. However, with the rules in Fig. 5, we can automatically generate constraints that are guaranteed to be satisfiable, thus acting as a ground truth for the compilation correctness. MR_{SIM} inserts constraints (the `assert` statements at lines 6, 9, and 12) into the source program. The constraints at line 6 and 9 are constructed with C-MAX and C-TAUT, respectively. Line 12 shows an example of forming more complex constraints by combining multiple rules. It first constructs two constraints with C-EQ and C-NLT with the profiled value of x (76). The two constraints are further combined with C-AND. Since all the constraints are constructed to be satisfiable, the compiled program should yield "SAT"; otherwise, it indicates a compiler bug.

D. MR_{IVM}: Information Visibility Mutation

Motivation. One of the key usages of ZK protocols, as the name "zero-knowledge" suggests, is to *preserve privacy*. It is critical to avoid private information being leaked to unintended parties while also allowing necessary public information to be readily available to support program execution. For instance, in a token mining application, different participants maintain two separate sets of tokens: public tokens, whose total amount can be visible to other participants, and private tokens, which should only be visible to the token owner [2].

In typical ZK programs, private and public variables are annotated with *private* and *public* types, respectively. Overall, information visibility imposes a major influence on the compilation pipeline: during the frontend analysis phase, ZK compilers need to perform visibility propagation (e.g., using taint analysis) to decide the visibility of all variables, including the ones that are transitively reachable from others [66], [91].

$$\boxed{\Gamma; V \vdash \mathcal{C}(e) \Downarrow v}$$

$$\frac{V(x) = v}{\Gamma; V \vdash \mathcal{C}(x == v) \Downarrow v'}\ \text{C-EQ}$$

$$\frac{V(x) = v}{\Gamma; V \vdash \mathcal{C}(\neg(x < v)) \Downarrow v'}\ \text{C-NLT}$$

$$\frac{\Gamma(x) = \{\nu : \varphi \mid \nu : \epsilon\}}{\Gamma; V \vdash \mathcal{C}(x \leq \text{MAX}(\varphi)) \Downarrow v}\ \text{C-MAX}$$

$$\frac{\Gamma(x) = \{\nu : \varphi \mid \nu : \epsilon\} \quad \varphi \in \{\texttt{int, uint, field}\}}{\Gamma; V \vdash \mathcal{C}(x == x \times 1) \Downarrow v}\ \text{C-TAUT}$$

$$\frac{\Gamma; V \vdash \mathcal{C}(e_1) \Downarrow v_1 \quad \Gamma; V \vdash \mathcal{C}(e_2) \Downarrow v_2}{\Gamma; V \vdash \mathcal{C}(e_1 \wedge e_2) \Downarrow v}\ \text{C-AND}$$

$$\frac{\Gamma; V \vdash \mathcal{C}(e_1) \Downarrow v}{\Gamma; V \vdash \mathcal{C}(e_1 \vee e_2) \Downarrow v'}\ \text{C-OR}$$

Fig. 5: Selected rules for generating constraints.

```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6        assert(x <= INT_MAX) // C-MAX
7      } else {
8        x -= 1;
9        assert(x == x * 1)  // C-TAUT
10     }
11   //      C-EQ  C-AND  C-NLT
12   assert(x == 76 && !(x < 76))
13 }
```

Fig. 6: An example of inserting constraints with MR_{SIM}.

In the backend lowering phase, private data computations are mapped to operations involving cryptographic primitives, thus principally preventing information leakage from the verifier; in contrast, public data are computed in a mundane way [91]. Thus, by performing the information visibility mutation (MR_{IVM}), we anticipate effectively stress-testing the ZK compilers' ability to correctly compile and optimize private and public data flows.

Implementation Challenge. Our observation shows that introducing more and complex interleaving between private and public data flows can increase the stress on ZK compilers. To this end, one may wonder about the possibility of directly mutating the visibility annotations (public vs. private) of local variables declared in a ZK program. However, we notice that ZK compilers do not allow users to directly specify the visibility of those local variables, as their visibility must be coherent with the visibility of variables they depend on (e.g., public/private inputs). Thus, directly mutating local variable visibility is infeasible.

Input Extraction-Based Visibility Mutation. It is clear that the visibility of a local variable should not change unless all of its

```
1 fn foo() {
2   let mut x = 81;
3   for i in 1..10
4     if x % i == 0 {
5       x += 1;
6     } else {
7       x -= 1;
8     }
9 }
```

(a) Original program with complex data flows.

```
1 fn foo(public u, private w) {
2   let mut x = 81;
3   for i in 1..10
4     if x % i == u { // 0 -> u
5       x += 1;
6     } else {
7       x -= w;        // 1 -> w
8     }
9 }
```

(b) Mutated program after extracting variables u, w as inputs.

Fig. 7: An example of information visibility mutation.

dependencies are simultaneously changed. This is challenging to achieve; we tentatively explored designing backward, static taint analysis to comprehensively track the dependencies of a local variable. We find that this approach is hard to scale and costly, mainly because we need to perform such a static analysis each time we mutate the visibility of a local variable.

Hence, we instead propose directly introducing extra program inputs to assist in changing the visibility of the local variables. Note that in this case, ZK compilers are forced to perform extra visibility propagation (conceptually similar to taint analysis) to decide the visibility of variables that are transitively reachable from the newly introduced program inputs. Formally, given a randomly selected expression e in the ZK program, MR_{IVM} introduces a new variable x to replace e:

$$\frac{\Gamma \vdash e : \{\nu : \varphi \mid \nu : \epsilon\} \quad V \vdash e \Downarrow v}{e[e \mapsto x] \quad I = I \cup \{x : \{\nu : \varphi \mid \nu : \epsilon'\}\} \quad I(x) = v}$$

, where we use I to represent the set of program inputs. The basic type φ of x is the same as e, and the visibility type ϵ' of x is randomly chosen to be either public or private. The input x is set to hold the same value as e during runtime.[3] Our intuition is that the more program inputs we introduce, the higher the chance that we can influence the visibility of the local variables (see below for an example). Note that the programs generated from the random program generator and MR_{SIM} do not contain any program inputs, and hence, we do not need to consider mutating the visibility of existing program inputs. We will evaluate the effectiveness of this strategy in Section VII-C.

Fig. 7 gives an example of MR_{IVM}. Fig. 7a shows the program before mutation. We omit the constraints inserted by MR_{SIM} for brevity. The function foo receives no inputs (we do not generate program inputs in the random program generation and the MR_{SIM} mutation stages). It is clear that the visibility of the local variable x is public because there is no private information in the program. In order to increase the complexity of the data flows, we introduce two new inputs u and w to the program, where the visibilities of the two inputs are randomly selected (u is public and w is private in this case). We randomly replace expressions in the original program with the newly introduced inputs and obtain the mutated program in Fig. 7b. In particular, we will replace constant expressions

[3]By doing so, we ensure that the constraints generated in the MR_{SIM} mutation stage are still satisfiable. In addition, the value of e is unique since the program after the MR_{SIM} mutation does not contain any program inputs.

TABLE I: Key hyper-parameters of MTZK.

Hyper-parameter	Default Value
Max #nested blocks	5
Max for-loop iterations	3
Max #statements	10
Max expression depth	2
Pr(*for-loops*)	0.2
Pr(*if-statements*)	0.2
Pr(*assignments*)	0.3

TABLE II: Overview of the evaluated ZK compilers.

Compiler	Commit	ZK Protocols	Application
ZoKrates	c7e4e2 (Sep 19)	SNARKs [51], [27], [52]	NightFall [13], [14]
Noir	91efe4 (Aug 30)	PLONK [44]	Aztec [9]
Cairo	0f0b37 (Oct 12)	STARKs [11]	Starknet [96]
Leo	9c20f4 (Sep 22)	SNARKs [51]	Aleo [3]

in the if-condition and the right-hand side of the assignment statement because those expressions are more likely to influence the data and control flow analysis for visibility propagation. After random substitution, the visibility analysis of x is much more complex than the original one, because x now depends on the public input u through indirect control-flow synergy effects through the for-loop at line 3 and an if-statement at line 4, while it is also influenced by a direct data assignment from a private input w at line 7. By random input substitution, we can effectively increase the complexity of the privacy-related control and data flows, thus stress-testing the ZK compiler.

VI. Implementation and Experiment Setup

Implementation. MTZK is implemented in about 3K LOC of Python code [110]. We allow users to configure seed generation through hyper-parameters, including the maximum number of statements and the probability of generating language features. We list representative hyper-parameters and their default values in Table I. The hyper-parameters we use during the experiment are also given in the artifact [110] to ensure the reproducibility of our results. The default values of these hyper-parameters are chosen based on our experience in writing ZK programs; we intentionally avoid excessively long compilation and execution time while also offering testing inputs with reasonable complexity. Also, we note that the default values of these hyper-parameters may not necessarily be optimal, and we recommend that users configure them to suit their needs better in practice.

Accommodating Different ZK DSLs. Our proposed IR aims to provide a unified representation of common features of mainstream ZK DSLs. With extra "glue code," the IR can be easily translated into the concrete syntax of the tested DSLs during the seed generation phase. We are also aware that different ZK DSLs may have unique features and requirements for the generated programs. For instance, the Noir DSL [64] does not support comparison between finite field numbers, while the ZoKrates DSL [105] can support such comparison with certain restrictions. We further provide generation configurations, e.g., allow_field_comparison, to accommodate the needs of different ZK DSLs.

TABLE III: Statistics of testing inputs.

Target	ZoKrates	Noir	Cairo	Leo
Avg. #Operations	130	247	302	122
Max #Operations	1,359	1,928	2,500	1,120
Avg. #EQC Gates	1,116	1,024	2,421	860
Max #EQC Gates	20,114	7,892	19,022	44,575

TABLE IV: Efficiency of MTZK.

Target	ZoKrates	Noir	Cairo	Leo
Total testing time	8 hr	115 hr	3 hr	4 hr
Gen. & mut. time	105 sec	220 sec	166 sec	241 sec
Time to find the first bug	6 min	30 min	13 min	2 min
Avg. #bugs per hour	0.9	0.06	0.6	0.5

TABLE V: Discovered compilation failures and logic errors.

Testing Target	ZoKrates	Noir	Cairo	Leo	Total
#Comp. Failures	1,089	3,747	119	1,761	6,716
#Logic Errors	1,504	962	675	250	3,391
Total #Errors	2,593	4,709	794	2,011	10,107

ZK compilers. We carefully review existing ZK compilers and choose the four most popular, easy-to-use, and well-maintained ZK compilers for our evaluation. We choose ZoKrates [104], Noir [64], Cairo [21], and Leo [4] as our testing targets. The details of the four testing targets are listed in Table II. All of the four compilers are industry-level ZK compilers that are widely used in real-world applications. For instance, ZoKrates has been used in NightFall [13], [14] to transfer tokens on the Ethereum blockchain; Noir is the default compiler in Aztec [9] with over 100M$ investment [1]. The compilers are also varied in their supported ZK protocols, covering SNARKs, STARKs, and PLONK.

Statistics of Testing Inputs. For each of our testing targets, we first generate 10K source programs, then mutate each with MR_{SIM} and MR_{IVM}, to obtain two mutants. As such, we have 30K testing inputs in total for each of our testing targets. We measure the maximum and average number of operations (e.g., additions and multiplications) in the testing inputs. We select the number of operations as the metric because ZK compilers typically compile the source programs into circuits, and the number of operations in the source programs is highly correlated with the complexity of the compiled circuits. We use the circuit inspection tool shipped by each compiler to measure the #EQC gates or the number of constraints in EQCs, depending on the EQC format and the utilities of EQC inspection tools. The statistics are in Table III. In general, a larger number of operations and constraints indicate a higher complexity of the testing inputs. We find that our testing inputs typically contain hundreds to thousands of operations, reflecting the complexity of our inputs, with the maximum number of operations reaching 2,500. The compiler-generated circuits are also highly complex and diverse, whose size ranges from a few hundred to dozens of thousands of gates. Overall, the statistics show that our testing inputs are highly diverse and complicated.

VII. Evaluation

We aim to answer the following research questions (RQs): **RQ1**: Can MTZK effectively and efficiently uncover errors in ZK compilers? **RQ2**: What root causes led to the failures of the tested ZK compilers? **RQ3**: How does each of the three testing stages contribute to error detection? **RQ4**: What are the potential security consequences of the uncovered compiler bugs?

A. *RQ1: Testing Efficiency and Effectiveness*

Testing Efficiency. All experiments are conducted on an Ubuntu 22.04 LTS server shipped with a 64-core Intel(R) Xeon(R) 6444Y CPU and 256G RAM. We empirically set the timeout to a sufficiently large value of 30 seconds (measured in real time). We use 32 threads to run MTZK in parallel. ZK compilers often launch multiple threads to compile and execute testing inputs, and therefore, we sum up the time spent in all the threads with the standard Linux utility `time`.

The statistics regarding the testing time of MTZK are shown in Table IV. The reported time is the sum of time spent on all threads. Overall, the testing input generation and mutation ("Gen. & mut. time") takes less than five minutes, while the total testing time takes several CPU hours to days, which is mostly spent on compiling, proving, and verifying ZK circuits. The significant time cost of ZK compilations is primarily due to the complex compilation pipeline involving extensive optimizations and transformations across different compiler IR layers. Also, proving and verifying ZK proofs is computationally expensive, as it often requires a large number of math operations, e.g., elliptic curve evaluations. As such, ZK programs are often much slower than normal programs (also justifying the need to continuously develop dedicated ZK compilers). The comparatively higher testing time on the Noir compiler can be attributed to its use of a different ZK protocol (PLONK), which has a higher cost of proving the circuits [44]. Thanks to parallelism, the real time to execute all testing inputs on Noir is within four hours.

MTZK is speedy at finding bugs. It finds the first bug within 2 to 30 minutes, demonstrating its efficiency in bug detection. During the entire testing campaign, MTZK discovers 0.5 bugs per hour across four compilers on average. MTZK has a lower bug discovery rate over the Noir compiler than the other three compilers due to Noir's high circuit proving overhead. The efficiency of MTZK mainly comes from its effective test case generation pipeline, which generates diverse and complex ZK programs (see Table III) that can stress-test the compilers. As discussed in Section IV-C, MTZK is a testing-based method. In line with other testing tools, MTZK does not guarantee that all bugs or a given specific bug will be found. Nonetheless, MTZK is efficient and can uncover bugs in ZK compilers in a timely manner. Also, as shown later in Section VII-B, MTZK offers nearly no false positives and uncovers error-triggering inputs that are helpful for developers to debug and patch bugs.

Discovered Errors. We report the number of discovered errors in Table V. Overall, MTZK finds a large number of errors in all of our testing targets, with hundreds to thousands of errors in each of the four compilers. We further break down

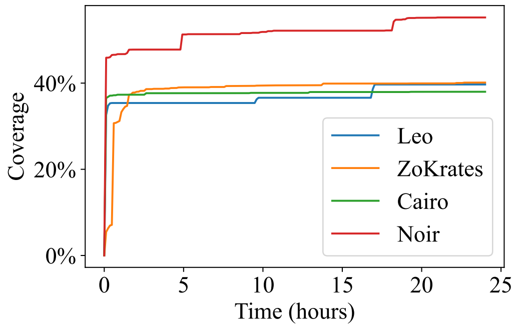


Fig. 8: Line coverage trend of the four ZK compilers.

TABLE VI: Testing results of RPG and EMI.

Target	RPG		EMI	
	#Err. Inp.	#Bugs	#Err. Inp.	#Bugs
ZoKrates	0	0	0	0
Noir	605	2	876	2
Cairo	598	2	22	2
Leo	0	0	0	0
Total	1,203	4	898	4

all the error-triggering inputs into compilation failures and logic errors. MTZK shows promising abilities in detecting both compilation failures and logic errors and is able to detect hundreds to thousands of both types of errors in all settings. MTZK typically uncovers more compilation failures than logic errors, which is expected because compilation failures are relatively easier to trigger. We find that compilation failures are normally caused by incorrect handling of the input programs, while logic errors are generally induced by subtle buggy optimizations and transformations. For instance, some logic errors in ZoKrates are only triggered when the value of a variable does not change in two consecutive iterations of a loop. Such error-triggering conditions are strict and hard to meet. In contrast, a compilation failure is triggered when the private inputs of a ZK program exceed the misconfigured bit width of the compiler. Such observation highlights the difficulty of detecting logic errors in ZK compilers. Still, MTZK detects over 3K logic errors in total, demonstrating its effectiveness.

Code Coverage. We launch the testing campaign for 24 hours and measure the line coverage of the tested ZK compilers. We exclude components that are not closely related to ZK compilation, e.g., the utility functions and debugging code. The coverage trend with regard to testing time is shown in Fig. 8. After 24 hours, MTZK achieves 39.7%, 40.1%, 38.0%, and 55.2% line coverage on the Leo, ZoKrates, Cairo, and Noir compilers, respectively. The coverage achieved on ZK compilers is already high, given that ZK compilers are developed with dozens of thousands of lines of code [105], [19] and contain both general-purpose and ZK-specific optimizations. The coverage on the Cairo compiler is relatively lower and only increases by 1.5% after 24 hours of testing. This can be attributed to the fact that the Cairo compiler has the largest code base among the four compilers, making it difficult to thoroughly test the compiler. Still, MTZK is seen to be successful in uncovering nearly 800 error-triggering inputs in the Cairo compiler.

Comparison with Baselines. MTZK is the first to systematically test ZK compilers, and we find no direct counterpart for comparison. Nevertheless, given the technical similarity between MTZK and traditional C compiler testing, we compare MTZK with two mainstream testing techniques: Random Program Generation (RPG) and EMI. RPG is a representative testing method to uncover compilation failures, while EMI employs exquisitely designed testing oracles to detect logic bugs in compilers. Both methods have uncovered thousands of bugs in production compilers [114], [98], [67]. We implemented both for ZK compilers. For EMI, we implemented the two most representative EMI mutations, i.e., dead code insertion [67] and live code mutation [98]. Since ZK programs have a unique execution paradigm enforcing simultaneous execution of both branches in an if-statement, dead code insertion requires special handling. As a workaround, we use ternary expressions to simulate dead code insertion of EMI.

We set up the comparison experiment with the same setting in Section VI for both RPG and EMI. The experiment results are shown in Table VI. Both methods uncover significantly fewer error-triggering inputs than MTZK. They uncover four bugs each, while MTZK uncovers 21 bugs in total. All the bugs uncovered by RPG and EMI are compilation failures. The inability of RPG and EMI to uncover logic bugs is due to the lack of constraints in their generated programs. Without constraints, a ZK program will trivially pass the verification and is not helpful in detecting logic bugs. Also, compiling ZK programs without constraints will be much easier, thus not stress-testing ZK compiler optimizations and undermining the overall bug detection capability. The inability of RPG and EMI to generate ZK-specific features highlights the need for the two ZK-oriented MRs of MTZK. By generating sophisticated constraints and data visibility, MTZK has a higher effectiveness in terms of overall bug discovery and is able to uncover both compilation failures and logic errors.

Answer to RQ1: MTZK successfully uncovers 10K error-triggering inputs in industry-level ZK compilers with high efficiency and good line coverage.

B. *RQ2: Bug Characteristics and Root Causes*

Error Deduplication and Bug Analysis. To understand the root causes of the uncovered error-triggering inputs, we manually debug and analyze the tested ZK compilers. Since multiple error-triggering inputs may be due to the same underlying bug, we first group the error-triggering inputs according to their erroneous behaviors and stages, and then analyze the root causes of each group to ensure the uniqueness of the bugs. In total, we obtain 34 groups of error-triggering inputs. Two authors with expertise in compiler design and ZK systems perform the manual analysis independently on all the groups and then discuss together to reach an agreement on the identified root causes. This step is costly, and it takes more than half a month to finish. We then report all of our findings to the ZK compiler developers to seek further confirmation.

False Positive Analysis. During the root cause analysis, we find several false positives (FPs), i.e., testing inputs that trigger errors in the compiler but are not due to bugs. Overall, we find less than 3% of FPs in the total error-triggering inputs. FPs are typically caused by early timeout (recall that we set the timeout to 30 seconds, but some testing inputs may take longer to finish). We also find a few FPs caused by insufficient stack size since our seed program generator also occupies part of the stack space. However, although these testing inputs do not uncover logic errors, the excessively long compilation and execution time or large stack size consumption could also be a problem for compilers; in certain scenarios, overly long processing time or resource consumption may cause DoS or so-called performance bugs [116], [61]; we leave it as future work to investigate such issues.

TABLE VII: Bug status.

Target	**ZoKrates**	**Noir**	**Cairo**	**Leo**	**Total**
Total Bugs	7	7	5	2	21
Confirmed Bugs	4	7	5	0	16
Fixed Bugs	3	7	5	0	15

Status and Distribution of Uncovered Bugs. Table VII shows the status of all the bugs we discovered. After deduplicating the 10K error-triggering inputs, we find 21 bugs in total. Through rigorous cross-checking from two experts and developer confirmation, all 21 bugs are distinct and are confirmed to be real bugs. ZoKrates and Noir have the highest number of bugs, with 7 bugs each. We have reported all the 21 bugs to the developers of the corresponding ZK compilers. The developers are highly responsive to our findings and immediately confirmed 16 of the reported bugs after receiving our reports. Moreover, 15 of 16 confirmed bugs have been fixed by the time of writing. The developers have assigned the rest of the bugs to the respective code writers for further investigation. We also characterize the distribution of our uncovered bugs in terms of the compilation pipeline. The distribution is illustrated in Table VIII. Overall, the bugs span across the whole compilation pipeline. The frontend and the circuit generation phases are seen to contain the highest number of bugs. The frontend performs a diverse set of optimizations, such as dead code elimination, whereas the circuit generation converts the low-level IR into EQCs. Thus, those two phases are highly complex and are prone to bugs.

TABLE VIII: Bug distribution in the compilation pipeline.

Target	**ZoKrates**	**Noir**	**Cairo**	**Leo**	**Total**
Frontend	2	4	2	2	10
Circuit Opt	0	3	2	0	5
EQC Gen	5	0	1	0	6

TABLE IX: Lines of code changes to fix bugs.

Target	**ZoKrates**	**Noir**	**Cairo**	**Leo**	**Agg.**
Avg. Patch Lines	451	522	71	NA	345
Max Patch Lines	858	1,489	106	NA	1,489

Bug Patch Complexity. We report the bug-fixing efforts in terms of the number of changed lines in Table IX. The "Agg." column means the average (maximum) number of lines of code changes across all the ZK compilers. Patching our uncovered bugs is seen to be a challenging task. In ZoKrates and Noir, the average lines of bug patches are around 500, with the maximum reaching 1,489. In addition, we find that all the compiler bugs require at least dozens of lines to fix. The level of patch complexity reflects the sophistication of ZK compilers. ZK compilers are typically highly complex, with extensive frontend and backend optimizations and transformations. As such, a seemingly small bug may be due to the synergy of multiple components in the compilation pipeline. For instance, a bug in the ZoKrates compiler can be triggered by one line of code. However, its bug fix patch takes over 300 lines of code to implement and spans across the frontend transformation and the circuit lowering components. The complexity of the bug patches also reflects the high quality of our uncovered bugs.

```
1 fn main() {
2   let a: u4 = -1;
3   assert(a == 15);
4 }
```

Fig. 9: Bug_1 example code.

Case Study. We present two representative bug examples, One of which is a logic error that leads to the rejection of a valid proof, while the other is a compiler crash. We also illustrate the root causes of the two bugs, respectively.

Bug_1: Incorrect Handling of Underflow. Fig. 9 presents a code snippet triggering a logic bug in the Noir compiler. Unlike traditional C programs, in which the behavior of integer underflow is undefined, ZK languages typically specify that after an integer underflow, the integer value becomes the maximum value of the integer type. In Fig. 9, the program first declares a 4-bit unsigned integer variable `a` at line 2. The variable `a` is assigned with `-1`, which, as a result, incurs an integer underflow. According to the semantics specification, the value of `a` should be 15. Thus, the constraint in line 3 asserting `a` is 15 should be satisfied (SAT). However, the proof is rejected during verification, indicating that the constraint in line 3 is unsatisfied (UNSAT). Such an issue is caused by a regression bug in the Single Static Assignment (SSA) transformation, and the value of `a` becomes the maximum value of the type `field` after the underflow. The Noir developer marks this as a high-priority bug [93], [63], and a patch is immediately released.

Bug_2: Erroneous Active Variable Analysis. Fig. 10 illustrates a bug in the Cairo compiler that causes a compiler crash. The source program declares a mutable variable `x` and initializes it to 1 in the first line. Since `x > 1` is true, the body of the if-statement at line 3 will be executed, and so will the loop at line 4. It can be easily checked that the if-condition at line 5 holds, and the loop will exit to enter another if-statement at line 11. Because `x` is still 1, the if-condition at line 11 will be true, and the constraint checking in line 12 asserting that `x == 1` should be SAT. Ideally, the execution of the generated circuit should yield an SAT result. However, the compiler crashes with an error message complaining about illegal access to a memory region. We find that the compiler crashes because of the erroneous active variable analysis. The compiler incorrectly marks `x` as an inactive variable after the loop at line 4, and thus the placeholder for `x` is not allocated in the circuit. As such, the constraint in line 12 asserting `x == 1` leads to illegal memory

TABLE X: #Bugs when applying testing stages separately.

Target	Prog Gen.	MR_{SIM}	MR_{IVM}	Total
ZoKrates	0	1	2	3
Noir	2	2	2	6
Cairo	2	1	NA	3
Leo	0	1	1	2
Total	4	5	5	14

TABLE XI: #Bugs when using three testing stages cumulatively.

Target	ZoKrates	Noir	Cairo	Leo	Total
Prog. Gen.	0	2	2	0	4
Prog. Gen.+MR_{SIM}	1	4	5	1	11
Prog. Gen.+MR_{SIM}+MR_{IVM}	7	7	5	2	21

access. Bug$_2$ can only be triggered from synergy effects of the loop and three if-statements, thus is challenging to detect.

Answer to RQ2: MTZK successfully uncovers 21 bugs, out of which 11 have been fixed. Such observation demonstrates the effectiveness of MTZK in finding ZK compiler bugs.

C. RQ3: Effectiveness of Each Testing Stage

To investigate the effectiveness of each testing stage, we apply each of the three testing stages separately. Table X shows the number of bugs discovered by each of the three testing stages. For MR_{SIM} and MR_{IVM}, we apply the corresponding mutations towards seed programs generated by random program generation ("Prog Gen."). For a fair comparison, we avoid seed programs that already trigger errors before the mutation. We do not perform information visibility mutation (MR_{IVM}) on the Cairo compiler because the compiler does not support information visibility annotations. Rather, privacy-related operations are handled by programmers in the upper-level ZK-based applications, such as Starknet [96], that use the Cairo compiler. Thus, we mark the corresponding entry as "NA" in Table X.

All three testing stages uncover a non-trivial number of bugs in the ZK compilers. Also, the three stages find a comparable number of bugs when applied individually. However, it does not necessarily mean that the three stages are equally effective in uncovering bugs. Random program generation can only uncover compilation failures since it does not generate constraints, which are essential to capture logic bugs in ZK compilers. The three testing phases, when used individually, uncover fewer bugs than when combined together (14 vs. 21). By inspecting additional bugs found by combining the three stages, we find that MR_{SIM} and MR_{IVM} achieve a plausible synergistic effect: without generating diverse and complex constraints from MR_{SIM}, mutating variable visibility alone (as done by MR_{IVM}) does not help to stress-test compilation for constraints; and without mutating variable visibility, MR_{SIM} alone cannot effectively alter the compiled EQCs.

We further show the combined effectiveness when using three testing stages cumulatively in Table XI. We find that adding more stages in the testing pipeline does not necessarily lead to a linear increase in the number of uncovered bugs: combining all the three stages finds nearly two times the number of bugs found by using only the first two stages. This observation suggests that the three stages are complementary to each other and can be used together to achieve a synergistic effect in practice.

TABLE XII: Potential exploitation types for uncovered bugs.

Exploitation Type	Denial-of-Service	Malicious User Inputs	Unapparent Exploitations
Bug Count	11	6	6

Answer to RQ3: All three testing stages are effective in uncovering errors in ZK compilers, and we recommend using all of them for a "synergistic effect" in practice.

D. RQ4: Security Implications of Bugs

Threat Model. We consider three threat models in our hazard analysis. Each of the three threat models considers a different type of adversary to exploit the uncovered bugs. We now introduce the threat model for each of the exploitation types.

Malicious User Inputs. This type of attack assumes that a malicious user learns that a target ZK compiler contains a critical logic bug, and confirms that the ZK application will be ill-compiled because of such a bug. By deliberately crafting malicious inputs, the attacker can construct proofs that should not be accepted by a correct verifier, but instead pass the verification due to the presence of the compiler bug. By cheating the ZK application, the user can gain unwanted privileges in an identity verification system [24], steal money by over-withdrawing from a bank or blockchain [57], or counterfeit a digital asset in a digital asset system [8], etc.

Denial-of-Service (DoS). This type of attack assumes that developers of ZK applications can send their source code written in ZK DSL to online deployment service providers [101], who will compile the source code with ZK compilers in the cloud. Due to the compiler bug, the compilation crashes the deployed service from the service providers, thus leading to denial-of-service attacks.

Hazard Analysis. Two authors of this paper first independently analyze the security consequences of each bug and construct exploits, then discuss together to reach an agreement. The bugs are categorized into three types according to their security implications, as shown in Table XII. The "Unapparent Exploitations" refers to the fact that the bug has no clear security consequences. The sum of all categories exceeds the total number of bugs because two of the "Malicious User Inputs" bugs can also lead to DoS attacks under certain conditions. For instance, one such bug in the Noir compiler can lead to stack overflow with complex control/data flow. In short, a large portion of the uncovered bugs are seen to have harmful consequences, especially the more severe ones like malicious user inputs. We further classify the six bugs in the "Malicious User Inputs" category into two types according to the exploitation methods. The first type requires inserting backdoors into the ZK applications, while the second type does not require any modification to the ZK applications. Three of

```
1  fn main() {
2    let mut x: u16 = 1;
3    if (x > 0) {
4      loop {
5        if x < 2 {
6          break;
7        }
8        x += 1;
9      };
10   }
11   if (x < 3 && x != 0) {
12     assert(x == 1);// crash
13   }
14 }
```

Fig. 10: Bug_2: Crash due to synergy of loop (line 4) with 3 if-statements (line 3, 5, 11).

```
1  fn low_income(pwd, salary){
2    let m = 12;
3    let avg_s = salary / m;
4
5    // Complex data flow
6    for ...
7    // Stealthily set m to 0
8
9    // Backdoor
10   if (pwd == MAGIC)
11     avg_s = salary / m;
12
13   assert(avg_s < 100);
14 }
```

Fig. 11: Backdoor attacking example where avg_s will be 0 when line 11 is triggered.

```
1 fn withdraw(account: &mut field, amount: field) {
2   let savings: field = read_savings(account);
3   assert(savings >= amount);
4   ... // Withdraw the money
5 }
```

Fig. 12: An example of malicious user inputs in a banking APP. When amount is set to the maximum value of the type field, the constraint in line 3 will always be satisfied.

the six bugs are of the first type, and the other three are of the second. We now give an example for each of the two types.[4]

Backdoor Exploitation of Divide-by-Zero. Fig. 11 shows a backdoor exploiting the divide-by-zero bug in the compiler. Assume a malicious developer can develop or modify the ZK application source code, e.g., by being hired as an outsourcing contributor or by committing code to the public repository.[5] The ZK application in Fig. 11 checks whether a person is a low-incomer. It takes the password pwd and the yearly salary salary of the checked person (line 1), and the verification will output SAT if the average monthly salary is less than 100 (line 13).A malicious developer crafts a division statement at line 11 that is disguised as a normal average calculation. The denominator m is secretly set to 0 through a (possibly obfuscated) complex data flow. As such, executing line 11 will stealthily assign avg_s with 0. Such a statement is further guarded by an if-statement at line 10 so that the backdoor can only be triggered when the password is set to a pre-defined value (i.e., MAGIC) that is only known to the developer. After the application is delivered to the contractor, e.g., the government, and the malicious developer becomes a user of the application, and can pass the low-incomer checking by triggering the backdoor with the pre-defined password MAGIC.

Bypassing Comparision Constraints. Fig. 12 shows an exploitation of an ill-compiled comparison constraint. A banking application uses ZK to verify whether a user has enough savings to withdraw a certain amount of money. The application first reads the current savings of the user account (line 2), and checks whether the requested withdrawal amount is within the savings (line 3). However, due to a compiler bug, the constraint checking in line 3 is ill-compiled, and when amount is set to the maximum value of the type field, the constraint checking will always be satisfied. As such, it is indicated that the user can withdraw any amount of money from the bank, even if the user does not have enough savings.

Answer to RQ4: MTZK uncovers many bugs that can lead to security consequences, including denial-of-service attacks, and more severe attacks like malicious user inputs.

VIII. Discussion

Limitations of Seed Quality. Our seed generation does not cover all features of the ZK compilers under test. For instance, we do not generate arrays and structs, and we do not consider advanced variable ownership features such as borrowing and references [18]. We hypothesize that arrays and structs may not help much in finding bugs, as the compiler will flatten them into a sequence of variables [81]. Considering advanced variable ownership features in code generation can be challenging [99], and not all ZK compilers support such features. We leave improving seed generation as future work, and we believe the current seed generation is sufficient to uncover a large number of bugs in ZK compilers.

Alternative Testing Methods. Since there are many ZK compilers in the industry and academia, one may question the feasibility of launching a differential testing approach [85], [56] by feeding identical inputs to different ZK compilers and checking their output consistency. Despite the simplicity of the idea, our preliminary study illustrates that such differential testing settings can be problematic. The reasons are two-fold.

Different Requirements on the Input Program. Different ZK compilers may have different requirements for their input programs. For instance, the Noir compiler requires that the input program cannot contain the comparison between finite field numbers, while the ZoKrates compiler can support such comparison when at least one of the operands is not a constant. As a result, the same input program may be rejected by one compiler but accepted by another compiler.

Different Runtime Behaviors. ZK compilers may use different ZK protocols as the runtime backend. For instance, Cairo-compiled programs are executed under the zk-STARKs protocol, while Noir-compiled programs are executed under the PLONK protocol. Different ZK protocols have incomparable semantics modeling the same ZK program. As a simple example, the elliptic curve for finite field arithmetic in the Cairo compiler is different from that in Noir. As a result, even though the same input program is executed with the same input under the two compilers, the execution result will be different (but both are technically correct) due to the distinct underlying ZK protocols.

Feedback-Driven Testing. MTZK does not use any feedback to guide test case generation. Nonetheless, it is technically feasible to extend MTZK with feedback-driven testing techniques [71], [33]. We tentatively use the erroneous behaviors and stages of the ZK compilers as feedback to guide the test case

[4]We view it as not an appropriate time to disclose the details of the six bugs in this paper, as their security concerns have not yet been fully resolved by the time of writing. Fixing those bugs requires significant changes to the compiler, and developers are still doing internal testing on their bug patches. We will release all bug details on [110] once all security issues are resolved.

[5]This assumption is reasonably achievable in practice, as major open-source ZKP applications like Polygon [88] accept community contributions. Attackers may stealthily insert backdoors into the committed code, as in the case of the XZ attack [43].

TABLE XIII: Bug finding capability with erroneous stages and behaviors of ZK compilers as feedback.

Target	Without Feedback		With Feedback	
	#Err. Inp.	#Bugs	#Err. Inp.	#Bugs
ZoKrates	2,593	7	2,581	7
Noir	4,709	7	6,068	8
Cairo	794	5	591	5
Leo	2,011	2	1,548	2
Total	10,107	21	10,788	22

generation process. If the triggered error by a testing program has unseen behavior or is triggered by a different testing pipeline stage, we put the program back into the queue and mutate it further with two MRs. Table XIII compares the bug-finding capability of MTZK with and without feedback. Feedback guidance incurs negligible overhead and takes nearly identical time to complete the testing process as without feedback. In terms of the overall bug-finding capability, we observe a slight improvement when using feedback. Feedback-guided testing helps to uncover one more bug. The additional bug is a compilation failure in the Noir compiler. With feedback, MTZK shows comparable performance in finding error-triggering inputs as without feedback. The number of error-triggering inputs ("#Err. Inp.") on the Cairo, Leo, and ZoKrates compilers is slightly reduced with feedback, possibly due to that the feedback guidance helps to generate dissimilar error-triggering inputs. An exception is that more error-triggering inputs are found in the Noir compiler, which may be attributed to the additional bug found in the Noir compiler. Overall, the feedback guidance is seen to have a limited impact on MTZK's bug-finding capability. Note that interested users can further explore more advanced feedback-guidance strategies to improve the efficiency of MTZK, and we have released MTZK [110] to facilitate usage and extension.

IX. Related Work

Fuzzing. Fuzzing primarily targets finding crashes or memory vulnerabilities that cause *obvious* misbehavior. Fuzzing has been applied to various critical software systems, including operating system kernels [100], [39], [16], [29], web browsers [86], and TLS libraries [38], [94]. Fuzzing has also been applied to novel domains, such as physical simulators [112] and Reinforcement Learning systems [84]. However, without a clear oracle to determine the correctness of the output, fuzzing may not be suitable for testing ZKP compilers.

Compiler Testing. Compilers are essential software tools for translating programs written in high-level language to low-level executables. Traditional compilers like GCC, LLVM have been extensively tested, with thousands of bugs uncovered to date [114], [73], [98], [69], [67], [77], [78]. Just-in-time (JIT) compilers such as JVM and JavaScript engines have also been the focus of the research community [58], [12], [72], [50]. Other compilers like shader compilers, deep learning compilers, and Multi-Party Computation (MPC) compilers are fruitfully tested using metamorphic testing or differential testing as well [32], [113], [111], [70], [79], [92], [74]. Besides discovering correctness issues, testing is also applied to identify performance issues in compilers [76], [102], [103]. Our work is the first to systematically test ZK compilers using metamorphic testing, with a focus on uncovering logic bugs.

Quality Assurance for ZK. Our work focuses on testing the correctness of ZK *compilers*. Another line of work focuses on finding bugs in ZK *application programs*. Circomspect [30] is a static analyzer for linting common buggy patterns, such as unused variables, in Circom programs. Picus [87], Ecne [35], **QED**^{2} [83], and Soureshjani et al.'s work [95] aim at discovering under-constrained circuits in ZKP programs. Wen et al. [108] study common vulnerability patterns in ZKP circuits and propose a static analysis framework for automatically discovering those patterns. Furthermore, Liu et al. [75] propose a statically typed language to formally specify and check properties of ZK applications, and Coglio et al. [31] launch verification towards snarkVM circuits. However, all of the existing works only focus on bugs in the ZK *applications*. Even if a ZK application were bug-free, bugs in the underlying ZK compilers could still render the compiled application program insecure.[6] Some ZK compilers have explored verifying ZK compilers [10], [41], [82], [28], yet the verification is only performed partially on the compilation pipeline or over a subset of language features, due to the high complexity of compiler verification. Industrial ZK compilers (e.g., our tested compilers) are never comprehensively verified by those formal approaches. In contrast, MTZK tests ZK compilers in an end-to-end and practical manner; its seeds cover many language features, and it uncovers bugs from the whole pipeline of commercial ZK compilers, whose findings are responsibly and promptly patched by the developers. [40] and [36] assess the performance of ZK compilers, but do not study their correctness. Moreover, the benchmarking was done on a small set of programs. To the best of our knowledge, our work is the first to deliver systematic and automated testing towards the correctness of ZK compilers.

X. Conclusion

We present MTZK to test ZK compilers. MTZK consists of two novel MRs and a comprehensive testing pipeline to automatically generate and mutate ZK programs. Our evaluation of four industry ZK compilers uncovers 21 bugs, 15 of which have been fixed. We also perform a hazard analysis of the uncovered bugs and show possible exploitations. MTZK is useful for ZK compiler developers to test their compilers and subsequently prevents security breaches in ZK systems.

Acknowledgment

The HKUST authors were supported in part by a RGC GRF grant under the contract 16214723, RGC CRF grant under the contract C6015-23G, research fund provided by HSBC, and a Webank research fund WEB24EG01. We are grateful to the anonymous reviewers for their valuable comments.

References

[1] Lifted A. Aztec network raises $100m in series b funding. https://nfthorizon.io/aztec-network-raises-100m-in-series-b-funding/, 2022.
[2] aharshbe. Token-mining application in aleo. https://github.com/AleoHQ/workshop/blob/master/token/src/main.leo, 2023.
[3] Aleo. Aleo - fully private applications. https://aleo.org/, 2023.

[6]This conceptually correlates to the classic backdoor attack noted in Ken Thompson's Turning award lecture [106] — "Reflections on Trusting Trust."

[4] Aleo. Leo — zero-knowledge programming language. https://leo-lang.org/, 2023.

[5] Robert Andrew. 21st century technologies: Zero-knowledge proofs. https://citylife.capetown/uncategorized/21st-century-technologies-zero-knowledge-proofs/31487/, 2023.

[6] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[7] Aztec. $50,000 usd bug bounty. https://twitter.com/aztecnetwork/status/1453773879898378241, 2021.

[8] Aztec. Disclosure of recent vulnerabilities. https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities, 2021.

[9] Aztec. Aztec - ethereum, encrypted. https://aztec.network/, 2023.

[10] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 488–500, New York, NY, USA, 2012. Association for Computing Machinery.

[11] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

[12] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 351–364, New York, NY, USA, 2022. Association for Computing Machinery.

[13] EY Blockchain. Nightfall. https://github.com/EYBlockchain/nightfall, 2021.

[14] EY Blockchain. Nightfall3. https://github.com/EYBlockchain/nightfall_3, 2023.

[15] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive*, 2019.

[16] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *Network and Distributed System Security (NDSS) Symposium*, 2023.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, 2008.

[18] Cairo. Cairo - linear types. https://docs.cairo-lang.org/language_constructs/linear-types.html, 2023.

[19] Cairo. Cairo repo. https://github.com/starkware-libs/cairo/tree/main, 2023.

[20] Cairo. cairo/docs/security.md at main · starkware-libs/cairo · github. https://github.com/starkware-libs/cairo/blob/main/docs/SECURITY.md, 2023.

[21] Cario. Cairo - a stark-based turing-complete language for writing provable programs on blockchain. https://www.cairo-lang.org/, 2023.

[22] Chainlink. Chainlink — zero knowledge proofs. https://blog.chain.link/zero-knowledge-proof-projects/, 2022.

[23] Chainlink. What are zk-rollups (zero-knowledge rollups)? https://blog.chain.link/zero-knowledge-rollup/, 2023.

[24] Kyle Charbonnet. Missing checks to ensure 'zero' is less than snark_scalar_field. https://github.com/privacy-scaling-explorations/zk-kit/issues/23, 2022.

[25] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong ..., 1998.

[26] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 2020.

[27] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Paper 2019/1047, 2019. https://eprint.iacr.org/2019/1047.

[28] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. *Cryptology ePrint Archive*, 2021.

[29] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693. IEEE, 2021.

[30] circomspect. Circomspect. https://github.com/trailofbits/circomspect, 2023.

[31] Alessandro Coglio, Eric McCarthy, Eric Smith, Collin Chin, Pranav Gaddamadugu, and Michel Dellepere. Compositional formal verification of zero-knowledge circuits. Cryptology ePrint Archive, Paper 2023/1278, 2023. https://eprint.iacr.org/2023/1278.

[32] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[33] Emine Dumlu, Cemal Yilmaz, Myra B Cohen, and Adam Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 243–253, 2011.

[34] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, 2018.

[35] ecne. Ecne. https://github.com/franklynwang/EcneProject, 2023.

[36] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks. Cryptology ePrint Archive, Paper 2023/1503, 2023. https://eprint.iacr.org/2023/1503.

[37] Ethereum. Ethereum — zero knowledge proofs. https://ethereum.org/en/zero-knowledge-proofs/, 2023.

[38] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540, 2020.

[39] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. {ACTOR}:{Action-Guided} kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5003–5020, 2023.

[40] Anoma Foundation. Zkp (circuit) compiler shootout. https://github.com/anoma/zkp-compiler-shootout, 2023.

[41] Cédric Fournet, Chantal Keller, and Vincent Laporte. A certified compiler for verifiable computing. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 268–280, 2016.

[42] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, 1991.

[43] Andres Freund. [oss-security] backdoor in upstream xz/liblzma leading to ssh server compromise. https://lwn.net/ml/oss-security/20240329155126.kjjfduxw2yrlxgzm@awork3.anarazel.de/, 2024.

[44] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

[45] Oliver Gale. Zero-knowledge proofs: Use-cases for businesses. https://www.forbes.com/sites/forbesbusinesscouncil/2023/04/19/zero-knowledge-proofs-use-cases-for-businesses/?sh=1e7f6e5c58b8, 2023.

[46] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. Hyperfuzzer: An efficient hybrid fuzzer for virtual cpus. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 366–378, 2021.

[47] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. https://eprint.iacr.org/2021/1063.

[48] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[49] Nihal R Gowravaram. *Zero Knowledge Proofs and Applications to Financial Regulation*. PhD thesis, 2018.

[50] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

[51] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.

[52] Jens Groth and Mary Maller. Snarky signatures: minimal signatures of knowledge from simulation-extractable snarks. Cryptology ePrint Archive, Paper 2017/540, 2017. https://eprint.iacr.org/2017/540.

[53] Samuel Groß. A case study of javascriptcore and cve-2016-4622. http://www.phrack.org/issues/70/3.html#article, 2021.

[54] Samuel Groß. Exploiting logic bugs in javascript jit engines. http://www.phrack.org/issues/70/9.html#article, 2021.

[55] Alyssa Hertig. Zcash team reveals it fixed a catastrophic coin counterfeiting bug. https://www.coindesk.com/markets/2019/02/05/zcash-team-reveals-it-fixed-a-catastrophic-coin-counterfeiting-bug/, 2019.

[56] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[57] Immunefi. Polygon double-spend bugfix review — $2m bounty. https://medium.com/immunefi/polygon-double-spend-bug-fix-postmortem-2m-bounty-5a1db09db7f1, 2021.

[58] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 43–55, 2023.

[59] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4949–4965, 2023.

[60] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. Detecting transactional bugs in database engines via {Graph-Based} oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 397–417, 2023.

[61] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[62] Swihart Josh, Winston Benjamin, and Sean Bowe. Zcash counterfeiting vulnerability successfully remediated. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/, 2019.

[63] Jurarpereurs. Proving phase fails for operations on negative literals. https://github.com/noir-lang/noir/issues/2098, 2023.

[64] kevaundray, jfecher, and French Tom. Noir. https://noir-lang.org/, 2023.

[65] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[66] George Kushnir. Visibility declarations inside structs. https://github.com/noir-lang/noir/issues/2135, 2023.

[67] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[68] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[69] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 327–337, 2015.

[70] Bastien Lecoeur, Hasan Mohsin, and Alastair F Donaldson. Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1801–1825, 2023.

[71] Daniel Lehmann and Michael Pradel. Feedback-directed differential testing of interactive debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 610–620, 2018.

[72] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating jit compilers via compilation space exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 66–79, New York, NY, USA, 2023. Association for Computing Machinery.

[73] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. Boosting compiler testing by injecting real-world code. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024.

[74] Yichen Li, Dongwei Xiao, Zhibo Liu, Qi Pang, and Shuai Wang. Metamorphic testing of secure multi-party computation (mpc) compilers. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024.

[75] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. Certifying zero-knowledge circuits with refinement types. *Cryptology ePrint Archive*, 2023.

[76] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. Exploring missed optimizations in webassembly optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 436–448, New York, NY, USA, 2023. Association for Computing Machinery.

[77] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):196:1–196:25, 2020.

[78] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proceedings of the ACM on Programming Languages*, 7(PLDI):181:1826–181:1847, 2023.

[79] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 248–260, New York, NY, USA, 2023. Association for Computing Machinery.

[80] Ceyhun Onur and Arda Yurdakul. Electanon: A blockchain-based, anonymous, robust and scalable ranked-choice voting protocol. *arXiv preprint arXiv:2204.00057*, 2022.

[81] Alex Ozdemir, Fraser Brown, and Riad S Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266. IEEE, 2022.

[82] Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting (in a compiler for zero knowledge proofs). Cryptology ePrint Archive, Paper 2023/778, 2023. https://eprint.iacr.org/2023/778.

[83] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1510–1532, 2023.

[84] Qi Pang, Yuanyuan Yuan, and Shuai Wang. Mdpfuzz: testing models solving markov decision processes. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 378–390, New York, NY, USA, 2022. Association for Computing Machinery.

[85] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM.

[86] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Jing Tian, and Mathias Payer. {GLeeFuzz}: Fuzzing {WebGL} through error message guided mutation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1883–1899, 2023.

[87] picus. Picus. https://github.com/Veridise/Picus, 2023.

[88] Polygon. Web3, aggregated. https://polygon.technology/, 2024.

[89] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1140–1152, 2020.

[90] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682, 2020.

[91] Yuyang Sang, Ning Luo, Samuel Judson, Ben Chaimberg, Timos Antonopoulos, Xiao Wang, Ruzica Piskac, and Zhong Shao. Ou: Automating the parallelization of zero-knowledge protocols. Cryptology ePrint Archive, Paper 2023/657, 2023. https://eprint.iacr.org/2023/657.

[92] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 968–980, New York, NY, USA, 2021. Association for Computing Machinery.

[93] shuklaayush. Casting negative literals to unsigned doesn't work. https://github.com/noir-lang/noir/issues/2045, 2023.

[94] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1492–1504, 2016.

[95] Fatemeh Heidari Soureshjani, Mathias Hall-Andersen, MohammadMahdi Jahanara, Jeffrey Kam, Jan Gorzny, and Mohsen Ahmadvand. Automated analysis of halo2 circuits. *Cryptology ePrint Archive*, 2023.
[96] StarkWare. Starknet. https://starkware.co/starknet/, 2023.
[97] stealdrop. Stealthdrop: Anonymous airdrops using zk proofs. https://github.com/stealthdrop/stealthdrop/blob/main/README.md, 2023.
[98] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863, 2016.
[99] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. Syrust: Automatic testing of rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 899–913, New York, NY, USA, 2021. Association for Computing Machinery.
[100] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1630–1644, 2023.
[101] Remix Team. Remix - ethereum ide and community. https://remix-project.org/, 2023.
[102] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 977–989, 2022.
[103] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–709, 2022.
[104] Schaeffer Thibaut, Macesic Darko, Jacob Eberhardt, Stefan Tai, and Kuhnert Dennis. Zokrates - a toolbox for zksnarks on ethereum. https://zokrates.github.io/, 2023.
[105] Schaeffer Thibaut, Macesic Darko, Jacob Eberhardt, Stefan Tai, and Kuhnert Dennis. Zokrates - github. https://github.com/Zokrates/ZoKrates, 2023.
[106] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.
[107] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.
[108] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. *Cryptology ePrint Archive*, 2023.
[109] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.
[110] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. Research artifacts for mtzk. https://sites.google.com/view/mtzk, 2024.
[111] Dongwei Xiao, Zhibo Liu, and Shuai Wang. Metamorphic shader fusion for testing graphics shader compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2400–2412, 2023.
[112] Dongwei Xiao, Zhibo Liu, and Shuai Wang. Phyfu: Fuzzing modern physics simulation engines. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1579–1591, 2023.
[113] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–28, 2022.
[114] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.
[115] Michał Zalewski. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/, 2021.
[116] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 199–208. IEEE, 2012.
[117] zcash. Zcash: Privacy-protecting digital currency. https://z.cash/, 2023.

TABLE XIV: Bug comparison with existing works.

Existing Work	#Compiler Bugs	#Overlapping with MTZK
CODA [75]	0	NA
ZKAP [108]	0	NA
QED^2 [83]	0	NA

TABLE XV: Invalid program causes and their corresponding error message patterns.

Invalid Reason	Error Message Pattern
Divide-by-zero	"divisor of zero"
Modulo-by-zero	"remainder of zero"
Integer overflow	"add/sub/mul overflow"
Unconstraint inputs	"unconstrained variables"

APPENDIX

A. *Comparison of Bugs with Existing Works*

To provide a fair and comprehensive comparison, at this step, we carefully reviewed previous papers related to ZK circuit bug hunting. At our best efforts and based on the information provided in their papers, we try to deduce whether the flaws they have reported are related to ZK compiler bugs, as a number of their findings are not shipped with full details. Table XIV shows the number of uncovered ZK compiler bugs in existing works, as well as the number of overlapping bugs with MTZK. In short, at our best efforts, we confirm that none of them have found any ZK compiler bugs. The rationale behind this can be attributed to their distinct design goal of detecting bugs in ZK applications, rather than focusing on the ZK compiler itself.

B. *Security, Privacy, and Ethical Concerns*

MTZK is designed to test ZK compilers and uncover logic bugs in the ZK compilers. We follow the principle of *responsible disclosure* steps [20] recommended by the ZK compiler community, which includes reporting through private channels and giving the maintainers a reasonable time to fix the bugs before making them public. We also open-source MTZK to promote the reproducibility of this research work. Our tool is, however, *not* designed for malicious purposes, such as leveraging the discovered bugs to attack ZK systems. We emphasize that if any security vulnerabilities are discovered during the result reproduction process, they should be reported to the maintainers of the ZK compilers.