Cost-Effective Testing of MPC Compilers

SEBASTIAN WATZINGER, TU Wien, Austria
VALENTIN WÜSTHOLZ, Diligence Security, Austria
DEEPAK GARG, MPI-SWS, Germany
MARIA CHRISTAKIS, TU Wien, Austria

Secure multi-party computation (MPC) enables privacy-preserving computations using secret data, with applications ranging from health care and finance to machine learning and blockchains. MPC compilers translate high-level function descriptions to the low-level representations required for the actual execution, making them critical for both usability and scalability of MPC. However, these compilers may contain logic bugs that cause them to quietly produce wrong outputs, the consequences of which could be catastrophic given the sensitive applications of this technology. Testing MPC compilers in order to find these severe bugs is, therefore, paramount.

With only a single testing tool currently available (which is not publicly available in its entirety and has several serious limitations), this issue is far from resolved. In this paper, we present BabelFuzz, a cost-effective framework for testing MPC compilers. By introducing an expressive intermediate representation (IR) for its seed-program generation, BabelFuzz is able to support multiple compilers that use different input languages, while keeping the development effort of adding new targets low. Even better, this approach allows us to translate our IR to mainstream languages, which provides a powerful differential-testing oracle for highly efficient bug detection.

BabelFuzz not only found 27 new logic bugs across four MPC compilers, but it is also able to rediscover every fixed bug the previous state of the art in testing MPC compilers found.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: secure multi-party computation, compiler testing, differential testing, metamorphic testing

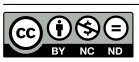
ACM Reference Format:
Sebastian Watzinger, Valentin Wüstholz, Deepak Garg, and Maria Christakis. 2026. Cost-Effective Testing of MPC Compilers. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE199 (July 2026), 22 pages. https://doi.org/10.1145/3808206

1 Introduction

Secure multi-party computation (MPC) is a class of cryptographic techniques that enable multiple parties to jointly perform privacy-preserving computations. In other words, each party should receive the result of a computation without, however, revealing its secret inputs to the other parties or being able to derive their secret inputs. As such, this technique has found several practical applications, e.g., in health care [18], financial fraud detection [33], user-data analysis [20], blockchains [8], and machine learning [57].

An MPC function is expressed as a high-level program in a domain-specific language (DSL) that is then compiled by an MPC compiler (e.g., [10, 11, 25]). The result is usually either an executable

Authors' Contact Information: Sebastian Watzinger, TU Wien, Vienna, Austria, sebastian.watzinger@tuwien.ac.at; Valentin Wüstholz, Diligence Security, Vienna, Austria, valentin.wustholz@diligence.security; Deepak Garg, MPI-SWS, Saarbrücken, Germany, dg@mpi-sws.org; Maria Christakis, TU Wien, Vienna, Austria, maria.christakis@tuwien.ac.at.



ACM 2994-970X/2026/7-ARTFSE199
https://doi.org/10.1145/3808206

or bytecode, which can be executed using specialized virtual machines (VMs). Unfortunately, there is no single standardized DSL, meaning that each compiler supports its own language, which is typically Python- or C-like.

Under the hood, the MPC compilation and execution pipeline is complex and highly optimized for runtime performance. In particular, intermediate representations such as circuits may be used, which aim to minimize the runtime overhead that is necessary to provide the expected security guarantees. To evaluate MPC programs, one of several cryptographic protocols is used; these differ significantly between each other and often provide slightly different security guarantees.

Due to their relative immaturity and additional complexity—in comparison to regular compilers—MPC compilers are all the more likely to contain bugs in their implementations. Moreover, the critical nature of MPC applications, i.e., involving confidential data, renders it imperative to develop techniques that validate the correctness of these compilers.

State of the art. Although formal verification of a compiler implementation is possible, it is overly costly given that it requires many person years to write thousands of lines of specifications [31]. As a result, there have emerged several automated testing techniques for compilers, which can detect bugs but not prove their absence; a recent survey on compiler testing [12] provides a comprehensive overview. These testing techniques may be grouped into the following categories.

First, *program generation* (e.g., [15, 19, 34, 35, 47, 50, 51, 56]) generates programs (either from scratch or via mutation) and checks whether a tested compiler crashes, times out, or triggers runtime errors when given these programs as input. Given the lack of a semantic oracle, these techniques primarily target robustness issues and typically miss more severe *miscompilation* bugs that lead to the successful generation of incorrect code. For this reason, program generation is often combined with the techniques described next.

Second, *differential testing* [39] (e.g., [14, 30, 44]) consists in running multiple compilers of the same source language on the same program. Miscompilation bugs are detected when the compiler outputs differ, e.g., in the emitted warnings or the behavior of the generated code. This technique is very practical as long as there exist at least two compilers that accept the same source language. Unfortunately, this is not the case for MPC compilers; they each compile their own DSL. Moreover, with differential testing, it is often unclear which compiler is to blame for differing outputs.

Third, *metamorphic testing* [13] (e.g., [17, 28, 29, 45]) typically transforms a given program such that the compiler output for the transformed program should not change (i.e., the semantics of the generated code should not change), or if it does change, the expected output should be known a priori. As an example, consider that a metamorphic transformation could introduce dead code in the original program to obtain the transformed program. Now, if the semantics of the generated code for these two programs differs, a miscompilation bug is detected. This example transformation is *equivalent*, and this type of transformation is the most commonly used one in metamorphic testing of compilers.

Despite the need to find bugs in MPC compilers, there has only emerged a single technique for testing them. It is implemented in a tool called MT-MPC [32], and it relies on both program generation and metamorphic testing. However, MT-MPC has the following main limitations: (1) the program-generation component is not publicly available, making MT-MPC impossible to execute, and the authors have not responded to our request for access; (2) based on its description, the program generation seems to be restricted to very few language features, primarily assignments for integer types, and may still produce programs with undefined behavior, such as division by zero; (3) the metamorphic transformations must be separately implemented for each supported DSL, which makes adding more transformations or compilers overly cumbersome.

Our approach. In this paper, we propose BabelFuzz, a tool and technique that combines program generation with both differential and metamorphic testing for MPC compilers—note that

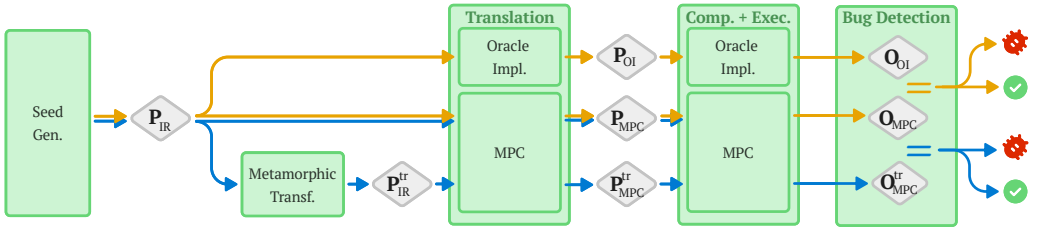


Fig. 1. Overview of BabelFuzz. Orange arrows: DT mode. Blue arrows: MT mode.

we enable differential testing for these compilers. In particular, our technique generates programs in a configurable and expressive intermediate representation (IR). By construction, the generated programs do not contain any undefined behavior since undefined behavior can lead to false positives during testing.

Next, the IR is translated to the different DSLs as well as to a mainstream, general-purpose language (Python in our case)—we refer to the latter as the *oracle implementation*. All translations are semantically equivalent, and the oracle implementation essentially specifies the correct behavior of the generated code. Therefore, BabelFuzz differentially tests each DSL implementation with the oracle implementation and detects a bug whenever there is a disagreement. Of course, the disagreement could arise due to a bug in the compiler or runtime of the mainstream language, however this is more unlikely than the bug being in an MPC compiler (in fact, it never happened during the course of this work). Consequently, the IR enables differential testing, and due to the additional oracle implementation, BabelFuzz can identify the MPC compiler at fault.

Through the IR, BabelFuzz makes metamorphic testing significantly more practical. In prior work, metamorphic transformations had to be implemented separately for each supported DSL, resulting in duplicated effort and limited scalability. In contrast, BabelFuzz applies transformations at the IR level, which can then be translated automatically to multiple target languages. Moreover, since differential testing with the oracle implementation provides a much stronger correctness oracle than metamorphic testing, the latter can now focus specifically on features that are difficult to model in the oracle implementation.

BabelFuzz primarily targets *logic bugs* in MPC compilers, consisting of miscompilation bugs and bugs in the underlying VM. Compiler crashes can also be found as a by-product, but we did not focus on them since they are less critical. We evaluate our technique by testing four popular MPC compilers, namely MP-SPDZ [4, 25], the EMP Toolkit [2], EzPC [3, 10], and Silph [6, 11]. BabelFuzz detected logic bugs in all compilers we tested; specifically, it found 27 unique bugs, 15 of which have already been fixed. We show that our tool is more effective than existing work and discuss our findings when comparing metamorphic testing with our variant of differential testing that relies on an oracle implementation.

Our contributions. Overall, our paper makes the following contributions:

- We present a general and cost-effective—both in terms of runtime cost for finding bugs and implementation effort—technique for testing MPC compilers for logic bugs.
- We implement our technique in the open-source tool BabelFuzz.
- We evaluate BabelFuzz by testing four popular MPC compilers. BabelFuzz detected bugs in all compilers; specifically, it found 27 unique bugs, 15 of which have already been fixed.

Outline. In Sect. 2, we give an overview of BabelFuzz, and in Sect. 3, we explain the technical details of our approach. Sect. 4 describes implementation aspects of BabelFuzz. We present our experimental evaluation in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.

```
1 def mpc_main():
2     inp = get_input(party=0, value=1,
      ↪ secret=True)
3     arr = init_array(size=1, secret=True)
4     arr[0] = inp * inp
5     arr[0] = 0 * 1
6     inp = inp * 2
7     return [inp, arr[0]]
```

(a) P_{IR} and P^{tr}_{IR}, represented as Python.

```
1 def mpc_main():
2     inp = sint.get_input_from(0)
3     arr = sint.Array(1)
4     arr[0] = inp * inp
5     arr[0] = sint(0) * sint(1)
6     inp = inp * sint(2)
7     return [inp.reveal(),
      ↪ arr[0].reveal()]
```

(b) P_{MPC} and P^{tr}_{MPC} in the MP-SPDZ DSL.

Fig. 2. Example for the workflow of BabelFuzz. The highlighted code represents code added by a metamorphic transformation.

2 Overview

In this section, we provide an overview of BabelFuzz, our framework for cost-effective testing of MPC compilers. We begin by outlining its high-level workflow, showing how BabelFuzz systematically generates, transforms, and validates test cases on compiler targets. We then introduce and elaborate on the two central ideas behind its design: (1) the use of a unified intermediate representation, which enables efficient and reusable test generation across different MPC DSLs; and (2) the integration of an oracle implementation, which allows for effective differential testing without requiring multiple MPC compilers.

2.1 BabelFuzz Workflow

Fig. 1 shows the high-level workflow of BabelFuzz. It supports two testing modes: one based on differential testing (DT) and one based on metamorphic testing (MT). The DT workflow consists of the following stages: seed generation, translation, compilation and execution, and bug detection. In MT mode, we additionally mutate the generated program using metamorphic transformations. In the following, we briefly describe each of these stages using an example that exposes a logic bug discovered by BabelFuzz in MP-SPDZ.

The seed-generation stage generates a program P_{IR} in a configurable and expressive intermediate representation; in particular, we use a Python abstract syntax tree (AST) to represent P_{IR}. By construction, P_{IR} is well defined. An example (presented in Python code) is shown in Fig. 2a (excluding the highlighted code). Line 2 defines a variable `inp`, which is a secret input of party 0. Next, we define a secret array of size 1, the first element of which is modified on lines 4–5. After another calculation (line 6), we return all currently visible variables and array elements on line 7, which results in `[2,0]`.

In MT mode, metamorphic transformations are applied on P_{IR} to obtain P^{tr}_{IR}. Fig. 2a (now including the highlighted code) shows an example, where a redundant multiplication is inserted on line 5.

Next, the translation stage translates P_{IR} (and P^{tr}_{IR} in MT mode) to a given target DSL. In DT mode, P_{IR} is also translated to a mainstream, general-purpose language—in our case Python. Recall that we refer to the latter as the oracle implementation. Note that the translation to Python is trivial since our IR is a Python AST. As shown in Fig. 1, we obtain a P_{MPC} program (Fig. 2b excluding the highlighted code), and either a P^{tr}_{MPC} program in MT mode (Fig. 2b including the highlighted code) or an oracle implementation P_{OI}, i.e., a Python program, in DT mode. P_{OI} is similar to our depiction of P_{IR} in Fig. 2a, except that the MPC-specific placeholders, such as `get_input`, are replaced by hard-coded values (e.g., `inp = 1` for line 2). All translated programs are semantically equivalent, and the oracle implementation constitutes a specification for the correct behavior of P_{MPC}.

After the respective programs are compiled and executed, the bug-detection stage checks for any disagreements in their outputs. In DT mode, we compare O_{OI} and O_{MPC}. BabelFuzz always assumes that the output of the oracle implementation is correct, and thus, any disagreement with this output is reported as a logic bug in the corresponding MPC compiler and execution environment. In our example, a bug in MP-SPDZ causes P_{MPC} to return `[2,1]` instead of `[2,0]` (as in the oracle implementation), and BabelFuzz reports this bug in DT mode.

In MT mode, on the other hand, we compare O_{MPC} (which is `[2,1]`) and O^{tr}_{MPC}. It so happens that our transformed program, P^{tr}_{MPC}, "avoids" the bug in MP-SPDZ, and consequently, O^{tr}_{MPC} is the correct output. As a result, BabelFuzz also discovers the bug in MT mode.

While DT mode is generally much faster at finding specific bugs—in this example, approximately 7x faster than MT mode (we discuss broader results in Sect. 5)—we include MT mode in BabelFuzz primarily because it may still be useful for validating certain language features that are difficult to model in our Python-based oracle implementation. Additionally, MT allows us to compare our differential-testing approach against traditional metamorphic testing in terms of both effectiveness and efficiency. This comparison is especially valuable since metamorphic testing, as implemented in MT-MPC [32], is currently the only existing approach for testing MPC compilers, although it is not fully publicly available.

2.2 Discussion on Cost-Effectiveness

The two key novelties of BabelFuzz are its use of an intermediate representation and of an oracle implementation.

Intermediate representation. Our technique generates well defined seeds in a configurable and expressive IR. First, our choice of IR—the Python AST—enables generating programs that are free from undefined behavior; see Sect. 3.2 for more details. In the presence of undefined behavior, compilers may emit executable code that runs in unpredictable ways, potentially leading to false positives during testing.

Second, the IR enables differential testing of compilers supporting diverse DSLs, e.g., the DSL of MP-SPDZ is Python-like whereas those of Silph and EzPC are C-like. Third, it also facilitates metamorphic testing since the transformations are applied on the IR and do not need to be separately implemented for each DSL. In fact, adding support for a new compiler requires comparatively little development effort. The necessary code for our currently supported compilers amounts to 280–400 lines of Python as developers merely need to add a translation to the compiler-specific DSL as well as a simple interface to access compilation and execution functionalities.

The expressiveness of the IR allows testing compilers with richer language features, e.g., Silph supports using secret inputs in conditionals or as array indexes, whereas EzPC does not. In addition, its configurability makes it possible to adjust the language features used during seed generation based on those that are supported by the tested compiler.

Oracle implementation. The oracle implementation has three important advantages. First, the main language features in popular MPC DSLs, e.g., arithmetic operations over integers, conditionals, loops, or arrays, are also present in mainstream imperative languages, such as Python or C. Consequently, translating to an oracle implementation does not compromise the expressiveness of the generated seeds.

Second, the oracle implementation enables differentially comparing the output of the MPC program to its (most likely) correct output. Of course, the output of the oracle implementation might be incorrect due to a bug in the compiler or runtime of the mainstream language, however this is far more unlikely than the bug being in the less mature MPC compilers. Comparing with the oracle implementation addresses two main limitations of standard differential testing. First,

standard differential testing requires more than a single target under test; in our case, BabelFuzz is able to test a single MPC compiler by comparing it with the oracle implementation. Second, with standard differential testing, it is typically difficult to tell which implementation is buggy when discovering a failing test. In our case, we can blame the MPC compiler with high confidence.

Finally, comparing with the oracle implementation comes at a much lower performance cost since compiling and executing Python or C is significantly faster than an MPC DSL.

Both our IR and oracle implementation are not inherently specific to testing MPC compilers. BabelFuzz is an instantiation of a more general testing workflow that can be applied to discover bugs in a wide range of compilers. Importantly, our approach addresses key limitations of traditional differential testing by providing a strong correctness oracle—particularly valuable in domains where target compilers accept diverse input formats that share a common set of core features.

3 Our Approach

We now describe BabelFuzz in detail. Before walking through its five stages, we first provide a brief overview of the intermediate representation at the core of our design. Finally, we discuss the potential for false positives and false negatives in our testing process.

3.1 Intermediate Representation

Recall that BabelFuzz uses Python ASTs as its IR. This choice has two main advantages. First, Python ASTs can be trivially translated to the oracle implementation, i.e., Python, enabling executable reference semantics. Second, they are easy to traverse and mutate facilitating both metamorphic transformations and translation to target MPC DSLs.

While Python ASTs can represent full Python, BabelFuzz intentionally restricts the IR to a well defined subset of Python constructs that are shared by the MPC DSLs we target and that can be translated reliably across backends. We summarize this subset below; its restrictions are enforced during seed generation and validated dynamically at runtime (Sect. 3.2).

IR constructs. The IR supports assignments of integer expressions to variables and one-dimensional arrays, as well as control-flow constructs, i.e., (nested) conditionals, bounded loops with a configurable maximum number of iterations, and jumps. It includes the following binary operators over integers: `+`, `-`, `*`, `//` (integer division), `<<` (left shift), `>>` (right shift), `**` (power), `%` (modulo), `&` (bitwise and), `|` (bitwise or), `^` (bitwise xor), `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&` (Boolean and), and `||` (Boolean or). This set captures the core language features shared by most MPC DSLs.

The IR does not currently include certain constructs provided by some MPC compilers. These omissions fall into three categories. First, some features are not yet supported but are readily addable within the IR, such as multidimensional arrays, user-defined functions, and richer aggregate data types. Second, fixed-point arithmetic introduces additional complications, as MPC compilers may incur precision loss or non-determinism. Accordingly, the IR is centered on integer computations: fixed-point types are introduced during translation only when available in the target compiler (Sect. 4). Third, pointer arithmetic cannot be faithfully modeled in Python; this constraint reflects a limitation of the oracle implementation rather than of the IR itself.

MPC-specific placeholders. To make MPC-specific concepts explicit while keeping the IR easy to translate, the IR represents them using a small, fixed set of placeholder functions embedded in the Python AST. These placeholders are limited to: (i) party inputs (`get_input`), (ii) initialization of integer variables with an explicit secrecy type (`init_int`), and (iii) array initialization (`init_array`).

Each placeholder function carries the MPC-relevant metadata required for translation. Concretely, `get_input(party, value, secret)` denotes a variable whose value is provided by party

Alg. 1: Seed-generation stage of BabelFuzz.

```
1  function GenerateSeeds(config):
2      inps := GenerateInputs(config);
3      state := GenerateInputValues(inps, config);
4      P_IR := InitSnippet(inps, state);
5      while ¬LimitReached(P_IR, config) do
6          P'_IR := GenerateSnippet(state, config);
7          valid, state' := RunSnippet(P'_IR, state, config);
8          if valid then
9              P_IR := P_IR + P'_IR;
10             state := state';
11     P_IR := FinalizeSnippet(state);
12     return P_IR;
```

`party` with secrecy `secret` and value `value`. Similarly, `init_int(value, secret)` denotes the initialization of an integer variable with value `value` and secrecy `secret`. Finally, `init_array(size, secret, values, secret_idx)` denotes a one-dimensional array of length `size` whose elements have secrecy `secret`; `values`, when present, provides initialization values, and `secret_idx` indicates whether the array may be indexed by secret expressions.

During translation, these placeholders are mapped to compiler-specific APIs. For example, in MP-SPDZ, the secrecy and party information encoded in `get_input` determine both the variable type (e.g., `sint`) and the party providing the input, resulting in code such as `sint.get_input_from(0)` in Fig. 2. In contrast, in the oracle implementation, placeholders are resolved by just producing the concrete value(s) specified by their parameters.

3.2 Seed Generation

The first stage of BabelFuzz generates seed programs in our IR and needs to meet two requirements. First, it should be highly configurable to cater to the diverse restrictions of MPC DSLs. For example, EzPC does not allow secret variables (that depend on secret inputs) in conditionals, whereas other DSLs do. A configurable seed-generation stage also enables guiding the testing effort, e.g., by controlling the program size, expression depth, frequency of used language features (for instance, ones that are particularly expensive or complex), value ranges, number of inputs, etc.

Second, this stage should not generate programs that raise errors in our oracle implementation. More specifically, the generated programs should not contain out-of-bounds array accesses and division- or modulo-by-zero operations. Additional care is needed to avoid behavior that is undefined for MPC compilers but not for our oracle implementation, as this may lead to false positives during testing. For instance, although integers do not normally overflow in Python, they do so in MPC DSLs. As another example, MP-SPDZ requires the bit length of division operands to be within a certain range.

BabelFuzz generates a well defined program, P_{IR}, as shown in Alg. 1. It starts by generating a number of inputs for each party together with their corresponding values and secrecy types, which are stored in *state* (lines 2–3). It then initializes the program snippet P_{IR} with these inputs (line 4). Next, the algorithm iteratively extends P_{IR} until a configuration limit is reached (line 5). Function GenerateSnippet on line 6 generates a random code snippet P'_{IR} while potentially using in-scope variables. In particular, GenerateSnippet tracks properties introduced by MPC-specific placeholders, such as variable secrecy, array sizes, and whether secret-dependent indexing is

Tab. 1. Metamorphic transformations in BabelFuzz.

Category	Transformation Description	Example		Source
Arithmetic expressions	Introduce algebraic identities that preserve semantics.	`- x = e`	`+ x = e * 1`	e.g., [17, 23]
Arrays	Lift scalar variable into array and redirect accesses to a fixed index.	`- x = 42` `- y = x * x`	`+ a[0] = 42` `+ y = a[0] * a[0]`	[17]

	Replace scalar variable with array of replicated values and redirect reads and writes accordingly.	`- x = 42`
		`- y = x + x`
		`- x = 123`
			`+ a = [42, 42, 42]`
			`+ y = a[1] + a[0]`
			`+ a = [123] * len(a)`
				original

Loops	Wrap statements in loop that executes exactly once (x = 1).	`- statements`	`+ for i in range(x):` `+  statements`	original

	Wrap statements in loop with many iterations but execute them once (x > 0, y ∈ [0, x − 1]).	`- statements`	`+ for i in range(x):`
			`+  if i == y:`
			`+  statements`
				original

	Wrap idempotent statements in loop with many iterations (x > 0).	`- statements`	`+ for i in range(x):` `+  statements`	original
	Insert dead loop with random body (x ≤ 0).		`+ for i in range(x):` `+  statements`	original
Dead jumps	Insert dead `break` in loops (`F` evaluates to `False`).		`+ if F:` `+  break`	[17]
	Insert dead `continue` in loops (`F` evaluates to `False`).		`+ if F:` `+  continue`	[17]
	Insert dead `return` of random output `o` (`F` evaluates to `False`).		`+ if F:` `+  return o`	[17]

permitted, and uses this information to constrain subsequent code generation. For example, array accesses are guided by the tracked array size, and secrecy-related restrictions of the target DSLs are respected.

Function RunSnippet (line 7) translates P'_{IR} to the oracle implementation and instruments it with runtime checks that detect undefined behavior for the tested compiler. These checks include, for example, bounded-integer overflow, loop-bound violations, invalid bit shifts, and division- or modulo-by-zero. The instrumented snippet is then executed with the values stored in *state*. The function returns a flag indicating whether all checks passed, as well as the updated state of the in-scope variables (*state*′). If the extension is valid, it is committed to P_{IR} and *state* is updated (lines 8–10); otherwise, the extension is discarded. Finally, FinalizeSnippet adds a return statement containing all visible variables and array elements to P_{IR}, and P_{IR} is returned (lines 11–12).

3.3 Metamorphic Transformations

The second stage of BabelFuzz is only used in MT mode, and its goal is to apply equivalent metamorphic transformations to P_{IR}. We now provide an overview of our transformations, shown in Tab. 1, and motivate their suitability for the MPC domain. Each row in the table corresponds to one transformation. The first column groups transformations by program construct, and the

second column informally describes them. The third column illustrates each transformation by showing a fragment of the original IR program (left) and the corresponding transformed fragment (right). The last column indicates whether the transformation is used in prior work or introduced in BabelFuzz, and provides references where applicable.

Arithmetic expressions. These transformations transform an arithmetic expression such that the new expression is semantically equivalent to the original one; such transformations have already been successfully used to test compilers in several domains (e.g., [17, 23]). More specifically, consider the arithmetic expression `e` in program P_{IR}. We can replace `e` in the following ways without altering the semantics of P_{IR}:

```
e * 1        1 * e        e + 0        0 + e        e // 1       e >> 0
e << 0       e & -1       -1 & e       e | 0        0 | e        e ** 1
```

These transformations are suitable for testing MPC compilers for three main reasons. First, given that the above constants (i.e., 0, 1, and −1) are known at compile time, the transformations could trigger code-elimination optimizations. Second, the constants could be secret variables, in which case neither the compiler nor the runtime may eliminate the additional calculations. As a result, compilation and execution of the transformed program becomes significantly more complex than that of the original program. In addition, compilers such as Silph or EzPC, which implement a hybrid MPC approach, might even change the low-level circuit representations for affected parts of the program. Third, these transformations could in certain cases cause variable types to change. For example, consider transforming statement `v = 42` to `v = 42 * w`, where `w` is a public runtime variable with value 1. For MP-SPDZ, the type of `v` in the original statement is a standard integer with a value known at compile time, whereas its type in the transformed statement is a public integer with a runtime value.

Arrays. These transformations, one of which was originally proposed for testing graphics shader compilers [17], replace variables with array elements.

For instance, the first array transformation in Tab. 1 lifts a scalar variable into an array and redirects all subsequent accesses to a fixed array index, following a standard array-lifting transformation [17]. We additionally introduce a new array transformation, shown as the second array transformation in Tab. 1. This transformation replaces a scalar variable `x` with an array `a` of arbitrary size whose elements are all initialized to the value of `x`. Subsequent reads of `x` are replaced by reads from a randomly chosen array index, while writes to `x` are replaced by writes to all elements of `a`.

These transformations test array accesses in general, but when using secret array indexes, they particularly stress MPC compilers that support this feature, e.g., MP-SPDZ or Silph.

Loops. A metamorphic transformation from related work on testing C compilers [45] is to enclose existing program statements in redundant conditionals. This transformation is already used in MT-MPC [32] for testing MPC compilers. In BabelFuzz, we apply similar transformations that add loops instead of conditionals. Note that a conditional can simply be viewed as a special case of a loop.

Our first loop transformation in Tab. 1 wraps a sequence of statements in a loop that executes exactly once. The second transformation similarly introduces a loop with multiple iterations, but ensures that the enclosed statements are executed only once by guarding them with a conditional. For idempotent statements, i.e., statements whose repeated execution does not change the observable program state, the third transformation wraps them in a loop that executes repeatedly without altering program semantics. Finally, the last loop transformation inserts a dead loop whose body is never executed.

As before, we may use secret or public values in the added code, e.g., for the argument of the `range` function. Apart from complicating the program control flow, which stresses compilers in

general [45], these transformations are particularly suited for MPC compilers as they can trigger loop-specific optimizations. In addition, the transformation inserting dead loops triggers code-elimination optimizations when the number of loop iterations is known at compile time.

Dead jumps. These transformations add jumps, that is, `break`, `continue`, and `return` statements, which, however, are not executed: they are wrapped in a conditional whose guard is constructed to evaluate to `False` by construction, similarly to prior work [45]. Dead-jump transformations have also been successfully applied to test graphics shader compilers [17].

As before, these transformations may use secret values to construct the guard. Moreover, they complicate the program control flow and may trigger code-elimination optimizations.

3.4 Translation

The translation stage of BabelFuzz translates P_{IR} (and P_{IR}^{tr} in MT mode) to the DSL of the tested compiler. In DT mode, P_{IR} is also translated to the oracle implementation. When adding support for a new compiler, this is one of the two BabelFuzz stages that needs to be extended; the other stage is compilation and execution, which we discuss next.

Recall that the seed-generation stage already translates snippets of P_{IR} to instrumented Python code; the instrumentation is used to detect undefined behavior given specific program input values. For these values, it is guaranteed that P_{IR} is well defined. Therefore, in DT mode, the translation stage simply translates P_{IR} to the oracle implementation (i.e., Python) without any instrumentation.

The translation to each DSL is tailored to the corresponding compiler and interprets the MPC-specific placeholders defined in the IR accordingly. While adding support for diverse compilers, we faced several challenges in translating IR programs to semantically equivalent programs in their respective DSLs.

First, type-related restrictions vary widely between DSLs. For instance, the EMP Toolkit does not permit arithmetic operations between secret and public integers:

```
Integer x = Integer(32, 42); // secret integer, bitlength 32, value 42
int y = 1; // public integer
Integer z = x * y; // ERROR: * not defined between public and secret integers
```

To address this, we cast the public operand to a secret type before applying the operation:

```
Integer z = x * Integer(32, y); // OK
```

Similarly, EzPC often requires explicit type annotations for operations involving mixed types:

```
// x, z are secret variables of type int32_al (arithmetically shared 32-bit
    ↪ integers)
// y is a public variable of type int32_pl (32-bit integer)
x + (z << y); // ERROR: (z << y) produces a boolean-shared result
```

We address this by casting the intermediate result back to an arithmetic share:

```
x + _al(z << y); // OK
```

Second, loop semantics differs across DSLs. In Python-like DSLs such as MP-SPDZ, modifying the loop bound within the loop body does not affect the number of iterations, whereas in C-like DSLs (e.g., EMP Toolkit), it does. This requires BabelFuzz to analyze loop bodies and, when necessary, generate helper variables to simulate consistent semantics. Furthermore, MP-SPDZ provides multiple loop types, but not all support `break` statements. BabelFuzz must select a loop construct compatible with the intended semantics of the IR program.

Third, integer division involving negative numbers is handled inconsistently across languages. Python rounds division results down, whereas C-like DSLs round toward zero. To maintain consistency, we modify the oracle implementation to use expressions such as `int(a/b)` instead of `a//b`, ensuring that rounding behavior aligns with that of the target compiler.

Fourth, some backends impose additional constraints on expressions or syntax. For example, MP-SPDZ lacks native bitwise operations on signed integers. To handle this, we use helper functions such as:

```
def and_bit_op(l, r, bitlen):
  result_bits = [a & b for a, b in zip(l.bit_decompose(bitlen),
    ↪ r.bit_decompose(bitlen))]
  return sint.bit_compose(result_bits) - result_bits[-1] * 2 ** (bitlen)
```

This approach decomposes the input integers into bit representations, applies the bitwise operation, and then re-composes the result, adjusting for signed values as needed.

Fifth, array-access semantics vary across compilers. For example, Silph does not allow array indexes to be derived from other array elements:

```
int c = a[b[0]]; // ERROR
```

BabelFuzz addresses this by rewriting the expression using a temporary variable:

```
int idx_a = b[0];
int c = a[idx_a]; // OK
```

While these adaptations address concrete syntactic and semantic differences between target languages, they are guided by a unified notion of semantic equivalence between the IR and its translations, which we formalize next.

IR-centric semantic equivalence. A key requirement of BabelFuzz is that programs translated from the IR to the target languages (either MPC DSLs or the oracle implementation) preserve its intended semantics. Because the IR is defined as a restricted subset of the Python AST, the oracle implementation preserves the IR semantics by construction. In particular, it resolves MPC-specific placeholders (e.g., inputs, secrecy annotations, and array initialization) and evaluates the resulting program under Python semantics, which serves as the operational semantics of the IR.

Definition 3.1 (IR-Centric Semantic Equivalence). Let P_{IR} be a program in BabelFuzz's IR, and let P_{MPC} be a program obtained by translating P_{IR} to a target MPC language. We say that P_{MPC} is *semantically equivalent* to P_{IR} if, for all concrete inputs admitted by the IR, executing P_{MPC} produces the same observable output as executing P_{IR} under the IR semantics.

Under this definition, the IR serves as the semantic reference model: all translations must preserve the IR-level computation, and semantic equivalence is defined relative to the behavior of the IR.

Translation contract. BabelFuzz does not attempt to translate arbitrary Python programs; instead, it translates only programs generated within its IR and admitted by the IR well-definedness checks. Semantic equivalence is ensured by construction through a translation contract that combines (i) restricted IR semantics, (ii) semantics-preserving translation rewrites, and (iii) oracle-based differential validation, which we explain next.

First, during seed generation, BabelFuzz restricts the IR to programs whose behavior is deterministic and well defined across the IR and all target languages. As described in Sect. 3.2, candidate IR snippets are dynamically executed under the IR semantics and rejected if they exhibit undefined or implementation-specific behavior (e.g., division or modulo by zero, invalid bit shifts, or DSL-specific constraints such as unsupported operand ranges). As a result, all IR programs admitted to the translation stage have a well-defined meaning under the IR semantics.

Second, target languages impose heterogeneous syntactic and semantic restrictions that are not present in the IR. To bridge these differences, BabelFuzz applies localized rewrites during translation that preserve the IR-level computation while adapting it to the constraints of the target language. These are described earlier in this section and do not alter the IR-level semantics.

Third, in DT mode, each translated DSL program is executed alongside the oracle implementation obtained from the same IR program. Any translation error that alters IR-level semantics would manifest as a mismatch between the oracle and DSL outputs and would therefore be detected by BabelFuzz during bug detection (Sect. 3.6). This serves as an additional practical safeguard against unsound translations.

3.5 Compilation and Execution

This stage of BabelFuzz compiles and executes the MPC program P_{MPC} as well as the oracle implementation P_{OI} in DT mode and the transformed MPC program P^{tr}_{MPC} in MT mode. Besides the translation stage, this is the only other stage of our tool that needs to be extended to support a new compiler.

In this stage, when compiling the MPC programs, BabelFuzz may optionally set random compiler flags, which should not affect the program outputs. For example, in the case of MP-SPDZ, a flag may activate dead-code elimination or specialized procedures for evaluating Boolean expressions, and in the case of Silph, it may modify the low-level representation of different program parts. This is common practice both in metamorphic and differential testing (e.g., [12, 28, 30, 45]).

3.6 Bug Detection

The final bug-detection stage of BabelFuzz compares the outputs of the program executions, i.e., O_{OI} and O_{MPC} in DT mode, or O_{MPC} and O^{tr}_{MPC} in MT mode. Note that we define a program output as the mutable state of the program (i.e., including variables and array elements) at the end of the main function. Since, in DT mode, BabelFuzz assumes that the output of the oracle implementation is correct, any differing output is reported as a logic bug in the MPC compiler and execution environment.

Each failing test case is manually verified before being reported as a bug. This process involves reducing the failing MPC program to a minimal example and determining whether the discrepancy between O_{OI} and O_{MPC} in DT mode (or between O_{MPC} and O^{tr}_{MPC} in MT mode) is truly caused by an MPC compiler or runtime error, rather than by an issue in BabelFuzz's implementation or in the oracle implementation.

3.7 False Positives and Negatives

As with any testing-based approach, BabelFuzz may in principle exhibit false positives or false negatives. We discuss these risks and how our design mitigates them.

False positives. A potential source of false positives is the generation of programs that exhibit undefined or implementation-specific behavior in MPC compilers. BabelFuzz mitigates this risk by restricting the IR to programs that are dynamically validated to be well defined under the IR semantics before translation (Sect. 3.2).

Another potential source of false positives is an incorrect implementation of the IR-to-target translation or of metamorphic transformations that do not preserve IR semantics. While BabelFuzz does not formally verify these components, such errors are expected to be detected in practice. First, translations are constrained by the IR-centric semantic equivalence definition (Sect. 3.4), and any translation error that affects IR-level semantics would be detected in DT mode. Second, incorrect metamorphic transformations would manifest as mismatches between the executions of the original and transformed programs in MT mode and would likewise be detected.

Each failing test case is manually validated before being reported as a bug. In our experience, this process has not revealed systematic translation or metamorphic-relation errors; instead, the detected discrepancies consistently correspond to real compiler or runtime bugs.

False negatives. False negatives are an inherent limitation of testing-based approaches, as not all bugs may be exercised by the generated test programs. BabelFuzz mitigates this risk by generating diverse IR programs, introducing dense data dependencies, applying a variety of metamorphic transformations, and leveraging both differential and metamorphic oracles.

4 Implementation

We highlight two interesting aspects of our implementation.

Fixed-point computations. As with mainstream programming languages, fixed-point computations in MPC DSLs may result in precision errors. To make matters worse, even for the same compiler, e.g., MP-SPDZ, such computations may be performed non-deterministically. For these reasons, fixed-point numbers have been completely avoided when previously testing MPC compilers [32].

In BabelFuzz, we provide limited support for fixed-point computations; we avoid the above issues by not generating computations that result in fractions or non-deterministic behavior. Concretely, we do not generate fixed-point computations with the division or modulo operators. As a result, it suffices to insert fixed-point types in the translation from our IR to the respective DSLs without modifying the seed generation. This is because the actual values of the fixed-point variables are integers, and thus, the Python oracle implementation can just use integer types. Clearly, this is a restrictive solution, but it allows us to test fixed-point types. In fact, it led to detecting three bugs in MP-SPDZ (see Sect. 5).

Swarm execution. In Sects. 2 and 3, we have described BabelFuzz starting from the generation of a single seed program. In practice, we of course run the tool continuously performing multiple such iterations. For each iteration, we generate a random configuration (*config* in Alg. 1) specifying various parameters of the testing procedure, such as the program size, number of inputs, etc. This loop terminates when a user-specified bound is reached.

5 Experimental Evaluation

We evaluate BabelFuzz by testing four popular MPC compilers, namely MP-SPDZ, the EMP Toolkit, EzPC, and Silph. In our evaluation, we address the following research questions:

RQ1: How effective is BabelFuzz in detecting logic bugs in diverse MPC compilers?
RQ2: What are characteristics of the detected bugs?
RQ3: How efficient is BabelFuzz?
RQ4: How does BabelFuzz compare to MT-MPC?

5.1 MPC Compiler Selection

MP-SPDZ [4, 25] is an actively maintained compiler and framework supporting more than 30 MPC protocols. It enables compilation from a Python-like DSL to custom bytecode, which can in turn be executed using virtual machines. MP-SPDZ implements many interesting features, such as accessing arrays using secret indexes, fixed-point computations, and limited hybrid MPC. Furthermore, it has several applications, such as privacy-preserving user-data analysis [20] and machine learning [26]. With around 1.1K stars on GitHub, it is the most popular compiler in our evaluation.

The **EMP Toolkit** [2] is a collection of C libraries implementing various MPC primitives using garbled circuits. We focus our testing efforts on the semi-honest two-party computation module. Contrary to other parts of the toolkit (which primarily implement MPC protocols), this module

offers convenient I/O functionalities for compilation and execution of MPC programs. Similar to MP-SPDZ, the EMP Toolkit also supports non-integer computations and has applications [8, 18].

EzPC [3, 10] is a compiler for two-party MPC that pioneered a hybrid compilation approach, generating protocols that automatically switch between arithmetic and binary representations. While this has the potential to improve the performance of the resulting executables, it also makes compilation significantly more complex. EzPC compiles a C-like DSL to C++ code that uses the ABY framework [16] to build its low-level circuits. This hybrid approach, along with the compiler's use in cutting-edge research on machine learning [1, 7], makes it an interesting testing target.

Silph [6, 11] is a relatively recent framework that also focuses on two-party hybrid MPC. To optimize performance, it automatically partitions the input program and chooses suitable low-level representations for the individual chunks using a wide variety of selection schemes. Silph compiles a subset of C to ABY bytecode, which can be executed using a custom interpreter. Interestingly, Silph hides almost all MPC-related considerations from users, who are merely required to specify the party providing each (secret) input of the main function. Unlike frameworks such as EzPC, it allows secret values in conditionals, which are translated to complicated multiplexer structures to avoid information leaking. Like MP-SPDZ, it also supports accessing arrays using secret indexes.

5.2 Experimental Setup

Testing procedure. We began applying BabelFuzz to MPC compilers early in its development and continued testing throughout its refinement. Our evaluation includes four compilers, each tested over a period of at least two months.

To maximize bug discovery efficiency, we halted testing on a compiler once a specific bug had been identified, preventing repeated rediscovery of the same issue. In cases where bugs remained unresolved for an extended time, we adapted BabelFuzz's configuration to avoid regenerating inputs that would trigger known issues, allowing us to focus on uncovering new ones.

With the exception of Silph, we tested the main branches of the respective repositories. Since the latest version of Silph is part of the CirC framework [42], we tested the CirC branch [6] suggested by the developers.

Fuzzing campaigns. BabelFuzz is configured to generate relatively small seeds (between 15 and 30 LOC when translated to Python), use a small number of loop iterations, and apply expensive operations (like divisions) less frequently. While BabelFuzz could generate seed programs for any number of parties, we limit our generated programs to two participants. For one, three out of the four tested compilers (EMP Toolkit, EzPC, and Silph) do not support more than two parties. In addition, previous work did not find the number of parties to be significant for their testing efforts [32].

For compilation and execution, we introduce a timeout of 200s. Furthermore, we choose comparatively fast (i.e., semi-honest) protocols for the actual execution of MP-SPDZ programs (since MP-SPDZ is the only framework we selected that allows users to choose MPC protocols).

In MT mode, we generate 5 transformed programs per seed program by stacking 10 to 15 metamorphic transformations. This has the advantage that the seed program only needs to be compiled and executed once in order to be used for 5 tests (i.e., comparisons to transformed programs), thus increasing throughput in MT mode.

In RQ3–4, we perform time-to-bug experiments, i.e., we measure the time it takes BabelFuzz to refind known bugs. For this, we focus only on bugs that have already been fixed and test the corresponding buggy versions of each compiler. This ensures that a detected failure corresponds to the intended bug: we automatically apply the developer-provided fix, rerun the failing program on the patched version, and check that the failure disappears. We run BabelFuzz with a 7-day time limit to keep the overall experiment schedule practical—given the large number of tested configurations

Tab. 2. Unique logic bugs detected by BabelFuzz.

Tool	ID	Status	Stage	Description
MP-SPDZ	1	Fixed	Exec.	Emulate VM: Comparisons of secrets
	2			Binary mode: Updating whole array
	3		Comp.	Array not updating
	4			Memory-safe array not updating
	5			Unexpected shallow copy of `sfix`
	6			Unexpected shallow copy of `cfix`
	7			Optimized loop accesses wrong array
	8			Binary mode: Wrong var updated
	9			Binary mode: Wrong update
	10			Loops not behaving as expected
	11			Unexpected NaN for fixed-point numbers
	12			Loop iteration skipped
	13			Small unrolling budgets impact result
	14			Large unrolling budgets impact result
	A	No Fix	Comp.	Bit shifts of negative numbers fail
	B			Division of negative numbers fails
EMP	15	Fixed	Exec.	Constructing negative secrets fails
	C	Conf.	-	Array updates in loops fail
Silph	D	Rep.	-	Division using secret numbers fails
	E			Comparing secret numbers fails
	F			Using `if` with secret guard
	G			Arrays of private and public elements
	H			Multiple bit shifts of secret values
	I			Modulo with negative constants
EzPC	J	Rep.	-	Signed types behave like unsigned
	K			Assigning secret array not working
	L			Modulo fails for arithmetic shares

and hardware-resource constraints—while still allowing the slower MT mode sufficient time to discover bugs. To guarantee reproducibility of our results, we repeat each fuzzing campaign with 10 random integer seeds and limit it to a single core.

Hardware. We ran all experiments on a Dell PowerEdge R6525 server with two AMD EPYC 7702 64-core processors and 2TB of RAM running Debian 12.

5.3 Experimental Results

We now discuss our findings for each research question.

RQ1: Effectiveness of BabelFuzz. To detect new bugs, we continuously ran BabelFuzz without any restrictions on time or resources (beyond our physical hardware limitations) using both DT and MT modes.

BabelFuzz discovered 27 unique bugs across all four MPC compilers. Of those, 15 have been fixed and 3 have been confirmed. An overview can be found in Tab. 2. The first column shows the affected MPC compiler. The second column provides the ID we assign to each bug; numbers indicate fixed bugs, whereas letters are assigned to the rest. Note that the IDs link to our anonymized bug reports

```
a = sfix(0)
b = sfix(a)
b.update(sfix(1))
print_ln('a: %s',
    ↪ a.reveal())
```

(a) Bug 5.

```
a = cint(2)
@for_range(a)
def _(i):
    a.update(i)
print_ln("%s", a)
```

(b) Bug 10.

```
a = sint(-1) >> sint(0)
print_ln('a: %s',
    ↪ a.reveal())
```

(c) Bug A.

Fig. 3. Logic bugs detected by BabelFuzz in MP-SPDZ.

on GitHub. The current status of the bug can be found in the third column (i.e., fixed, confirmed but no fix, confirmed, reported). For bugs that were fixed or for which we received an explanation from the developers, we provide the stage of the MPC pipeline where the bug occurs (i.e., compilation or execution). The last column contains a short description for each bug.

Although we tried our best to avoid reporting duplicates, we cannot entirely rule them out for unconfirmed bugs. Without a fix or comment from the developers, we cannot definitively determine whether multiple reported bugs share the same root cause. For such unconfirmed bugs, we reduce the likelihood of duplicates in practice by adjusting the testing configuration to avoid regenerating known triggering patterns once a bug has been identified.

While we used both DT and MT modes, bugs 2 and 10 are the only ones we initially discovered using MT.

All bugs reported to the developers of MP-SPDZ and EMP Toolkit have been either confirmed or fixed. The bugs in EzPC and Silph remain open, primarily due to limited bandwidth on the part of the respective developer teams. For Silph, the developers expressed interest via email, but have not yet provided fixes.

RQ2: Detected logic bugs. In this RQ, we discuss the causes of a few sample bugs that BabelFuzz detected. Note that we only consider bugs that were either fixed or explained by the respective developers.

Our running example from Sect. 2, specifically P_{MPC} in Fig. 2b (excluding the highlighted code), shows an MP-SPDZ program triggering bug 3; this bug was caused by faulty optimizations. On line 3, we construct an array `arr` of size 1 containing secret integers. On line 4, we set `arr[0]` to the result of a secret computation and directly after to 0 (line 5). The subsequent secret computation on line 6 is necessary to trigger the bug. Finally, we return the array element on line 7, which should have value 0. Instead, 1 is returned for the array element, suggesting that the second assignment to it (line 5) was ignored.

The reason for this miscompilation is a bug in the main optimization procedure of MP-SPDZ. Since the main bottleneck of secret-sharing-based MPC is the number of required communication rounds during execution, MP-SPDZ tries to batch independent operations that require communication [25]. To model this, a dependency graph is created. However, the compiler did not record consecutive write accesses to the same array element as being dependent on one another. This led to the array element not being updated right away, which caused calculations depending on it to return wrong results. While this issue was promptly fixed, BabelFuzz soon discovered a similar one (bug 4): when using memory-safe mode (which is meant to fix known issues with arrays[1] and can be activated via a compiler flag), different functions in the compiler were used for handling array accesses, which included the same bug as described above.

Another source of logic bugs were shallow copies, e.g., see bug 5 in Fig. 3a. Constructing a new secret fixed-point number (`sfix`) with a variable as argument on line 2 is meant to create a deep

[1] https://mp-spdz.readthedocs.io/en/v0.4.0/troubleshooting.html#order-of-memory-instructions-not-preserved

Tab. 3. Time-to-bug results for DT mode of BabelFuzz for our fixed bugs.

ID	Succ.	Min	Median	Max
1	10/10	0:01	**0:54**	7:00
2	10/10	4:34	**1:18:56**	2:25:10
3	10/10	10:41	**2:26:10**	4:24:13
4	10/10	40:21	**14:10:26**	1d18:22:24
5	10/10	0:06	**7:38**	39:19
6	10/10	4:44	**11:15**	18:30
7	0/10	-	-	-
8	10/10	4:50	**1:20:12**	5:25:18
9	10/10	12:48	**3:06:06**	19:14:28
10	10/10	4:58	**24:09**	1:22:10
11	1/10	-	**2d09:55:46**	-
12	0/10	-	-	-
13	1/10	-	**3d10:51:03**	-
14	0/10	-	-	-
15	10/10	0:15	**0:31**	02:25

Tab. 4. Time-to-bug results for MT mode of BabelFuzz for our fixed bugs.

ID	Succ.		Min	Median		Max
1	10/10	=	0:06	**6:33**	+628%	1:19:47
2	10/10	=	48:24	**12:10:51**	+826%	1d09:28:05
3	10/10	=	23:12	**17:12:33**	+606%	2d15:23:29
4	9/10	-1	6:29:36	**2d05:41:07**	+279%	3d13:07:51
5	10/10	=	1:54	**1:16:55**	+908%	7:09:25
6	10/10	=	1:10:00	**2:10:40**	+1,061%	3:53:24
7	0/10	=	-	-	-	-
8	10/10	=	14:09	**2:24:52**	+81%	1d01:57:37
9	8/10	-2	2:34:13	**2d02:01:22**	+1,513%	4d07:53:33
10	10/10	=	9:47	**4:18:11**	+969%	14:54:38
11	0/10	-1	-	-	-	-
12	0/10	=	-	-	-	-
13	0/10	-1	-	-	-	-
14	0/10	=	-	-	-	-
15	10/10	=	8:11	**44:35**	+8,529%	1:55:08

copy of `a` and store it in `b`. As a result, `a` and `b` should be independent variables that have the same value after line 2. However, bug 5 caused line 2 to make `a` and `b` point to the same value instead. Therefore, reassigning one of the two variables mutates the value of both.

Shallow copies can also lead to unintended behavior in other subtle ways, as bug 10 in Fig. 3b shows. In theory, loops in MP-SPDZ are meant to behave like Python loops. That is, the number of iterations should be fixed upon loop entry—the `range` function evaluates to a list, the length of which dictates the number of loop iterations (when ignoring `break` statements). Internally, however, MP-SPDZ checks the value of the argument to the `for_range` annotation at every iteration. This becomes a problem if this argument is a variable that is mutated inside the loop; MP-SPDZ does not copy its value upon loop entry, but only a reference to the value.

In the program of Fig. 3b, we assign a public runtime integer of value 2 (`cint(2)`) to variable `a` (line 1). Next, we enter a loop using `a` as the argument of the `range` function, for which MP-SPDZ uses the `for_range` annotation. Inside the loop (line 4), we set `a` to the current value of the loop counter `i`, which should have no effect on the number of loop iterations. However, since MP-SPDZ only stored a reference to `a` upon loop entry, it exits after one, instead of two, iterations, leading to a wrong result being printed on line 5.

Last, we discuss bugs caused by certain operations not being implemented, e.g., see bug A in Fig. 3c. In this case, bit shifts are not implemented for negative secret integers (since this is harder to do than for public ones). On line 1, we construct a secret integer with value -1 (`sint(-1)`) and shift it by 0 using another secret integer (`sint(0)`); the shift should have no effect at all. On line 2, we print the value of `a` and get $2^{63} - 1$, instead of -1. Interestingly, users are not warned about this behavior. The compiler cannot issue a warning since secret numbers are runtime values. In addition, the VM executing the actual protocol cannot crash since this would reveal information about the secrets in the computation.

Bug B reveals a similar issue. Public runtime integers (`cint`) are implicitly assumed to be unsigned, which is why certain operations, such as division, are not implemented for negative `cint`. Note that many other operations (such as addition, subtraction, and multiplication) work as expected, and the restriction to non-negative values is not documented at the time of writing[2].

[2]https://mp-spdz.readthedocs.io/en/v0.4.0/Compiler.html#Compiler.types.cint

RQ3: Efficiency of BabelFuzz. To evaluate the efficiency of BabelFuzz, we perform time-to-bug (TTB) experiments, i.e., we measure the time it takes to find fixed bugs (see Sect. 5.2 for the details of the setup).

Tabs. 3 and 4 present the results of the TTB experiments for DT and MT mode, respectively. The first column shows the bug ID, which is the same as in RQ1. The second column shows for how many of the random integer seeds the target bug was refound. The remaining columns present the respective minimum, median, and maximum TTB across all 10 runs. For experiments with just one successful run, only the median TTB is shown. Tab. 4, which presents the results for MT mode, also shows how the second and fourth columns compare to DT mode. For example, MT mode detected bug 9 with two random seeds less than DT mode and in 1,513% more median time.

Overall, BabelFuzz in DT mode detects 10 out of 15 bugs for all 10 random seeds, while bugs 11 and 13 are discovered for only one seed each. Bugs 7, 12, and 14 are not detected within the time limit. These infrequently detected bugs require highly specific circumstances to be triggered, which the seed-program generator does not reproduce within the allotted time. The median TTB for DT mode ranges from 31s to 3d11h. The success rate of MT mode is comparable to that of DT mode. The only differences are bugs 4, 11, and 13, each with one fewer successful run than in DT mode, and bug 9, which is detected for two fewer seeds. Furthermore, the median TTB in MT mode, ranging from 6m33s to 2d6h, is significantly higher than in DT mode for all bugs. This is not surprising as the throughput when testing, for example, MP-SPDZ in DT mode is 204% higher compared to MT mode (where one test corresponds to the comparison between the MPC program and the Python program in DT mode, and the comparison between a single transformed program and the seed program in MT mode).

This discrepancy in throughput is caused by the following. (1) The transformed programs used in MT mode are larger and more complex than the seeds (due to, e.g., inserted dead code), which leads to increased compilation and execution times. (2) In MT mode, we need to compile and execute the MPC version of both the seed and transformed programs. Recall that the results for the seed can be used for 5 tests (as we generate 5 transformed versions per seed to amortize costs); this brings the total required MPC compilations and executions per test down to 1.2. Given these numbers, we would expect an increase in TTB in MT mode of about 200% if both approaches were equally suitable to detect our logic bugs. However, for most bugs, the increase in TTB in MT mode is well above that threshold suggesting that DT mode is, in fact, preferable. Bug 8 represents the only exception. Here, MT mode only requires 81% more TTB, suggesting that it actually performs fewer tests until the bug is found.

Overall, these experiments show that DT mode is at least as powerful as MT mode. That is, all bugs that can be detected with MT mode can also be detected with DT mode. However, it is unclear if the opposite holds. For one, we only included fixed bugs in these experiments. It could, therefore, be the case that other bugs that DT mode found (and which have not yet been fixed) are not detectable with MT mode. The same applies to bugs 7, 12, and 14, which could not be refound by either mode. Furthermore, recall that DT mode found the vast majority of our bugs in RQ1; MT mode only discovered two, both of which can also be detected with DT mode.

As a secondary indicator of efficiency, we measured the relative amount of time BabelFuzz spends on its own procedures (i.e., seed generation, metamorphic transformations, etc.) compared to third-party calls (i.e., MPC compilation and execution). Our results show that, across all four tested compilers, BabelFuzz spends the vast majority of time (between 77% and 99%) on MPC procedures attesting to the efficiency of its own functionalities.

In summary, *BabelFuzz is able to consistently detect severe logic bugs, most within a time limit of 24h, with either mode (DT or MT). DT mode is more efficient and at least as powerful as MT mode.*

Tab. 5. Time-to-bug results for DT mode of BabelFuzz for bugs found by MT-MPC.

ID	Succ.	Min	Median	Max
RW1	10/10	0:01	**1:10**	5:02
RW2	10/10	0:43	**9:19**	18:59
RW3	10/10	3:00	**19:02**	53:31
RW4	10/10	1:40	**13:58**	43:59
RW5	10/10	7:36	**4:01:42**	11:40:48
RW6	10/10	0:01	**8:00**	49:42
RW7	10/10	1:05	**5:41**	10:42
RW8	10/10	11:45	**52:28**	2:59:59
RW9	10/10	0:14	**1:22:29**	6:37:42

Tab. 6. Time-to-bug results for MT mode of BabelFuzz for bugs found by MT-MPC.

ID	Succ.	Min	Median	Max
RW1	10/10 =	5:27	**12:47** +996%	34:52
RW2	10/10 =	8:46	**1:52:41** +1,109%	8:49:30
RW3	10/10 =	25:11	**2:54:14** +815%	15:17:23
RW4	10/10 =	18:31	**2:05:28** +798%	1d01:18:31
RW5	10/10 =	1:21:36	**20:57:42** +420%	5d22:47:46
RW6	10/10 =	1:22	**1:44:48** +1,210%	10:44:20
RW7	10/10 =	20:32	**2:02:18** +2,052%	3:25:37
RW8	10/10 =	2:59:02	**11:49:53** +1,253%	1d13:08:57
RW9	10/10 =	5:45	**17:21:23** +1,163%	2d06:36:40

RQ4: Comparison to MT-MPC. We compare BabelFuzz to MT-MPC [5, 32], which exclusively relies on metamorphic testing to find logic bugs in MP-SPDZ, the EMP Toolkit, and EzPC. Unfortunately, we cannot compare the tools directly; we do not have access to the seed-generation component of MT-MPC, and the authors did not reply to our request to release the missing parts of their tool. Instead, we employ the TTB experiment introduced in RQ3 to determine how many of the bugs originally discovered by MT-MPC can be found using BabelFuzz. We again only consider fixed bugs in this experiment. As before, we limit each experiment to 7 days and one core.

Tabs. 5 and 6 show the results of our TTB experiments for the fixed bugs originally detected by MT-MPC for DT and MT mode of BabelFuzz, respectively. The structure of the tables is similar to the ones presented in the previous RQ. Note that these bugs exclusively affect MP-SPDZ.

As our results show, *BabelFuzz can consistently discover all bugs of MT-MPC with a median TTB between 1m10s and 4h02m for DT mode. Most bugs are actually found within less than one hour.* As is the case of our own bugs (discussed in the previous RQ), MT mode is able to discover the same bugs as DT mode but performs significantly worse in terms of bug-finding time.

5.4 Threats to Validity

Our results depend on the MPC compilers under test as well as on the randomness of our seed generation and metamorphic transformations. To mitigate these threats, we chose a diverse set of compilers that employ distinct techniques, both in conception and implementation. Furthermore, we ran each of the structured experiments multiple times using different random integer seeds.

6 Related Work

In this section, we compare BabelFuzz with MT-MPC and discuss differential- and metamorphic-testing techniques in general. Note that we do not attempt to provide a comprehensive overview of compiler testing; see [12] for that.

MT-MPC. MT-MPC [32] is the only systematic-testing technique for MPC compilers and the most closely related work. It is exclusively based on metamorphic testing and implements three types of metamorphic transformations. The first type of transformations modifies data and control flow in an equivalent way, as was done in previous work [45], e.g., by inserting dead code, wrapping statements in redundant conditionals, etc. The second type changes data visibility, e.g., by making private variables public or vice versa. The last type changes the low-level circuit representation by either modifying variable data types or compiler flags.

Using its transformations, MT-MPC detected a total of 13 logic bugs across three MPC compilers: MP-SPDZ, EMP Toolkit, and EzPC, of which 9 were fixed. While some of BabelFuzz's metamorphic

transformations achieve similar effects—such as modifying control flow or altering low-level circuit representations—both tools support transformations that the other does not. Nevertheless, as shown in RQ4, BabelFuzz in MT mode is able to detect all bugs previously found by MT-MPC. More importantly, BabelFuzz's DT mode finds these bugs significantly faster: 7 out of 9 are discovered in under one hour, with 6 found in less than 20 minutes. In addition, BabelFuzz uncovered 21 new bugs in the same three compilers.

Apart from these results, BabelFuzz has the following benefits over MT-MPC. First, even though the seed-generation component of MT-MPC is not available, it seems to primarily generate assignments for integer types, and the programs may contain undefined behavior. In contrast, BabelFuzz generates much more diverse seeds, which are additionally well defined. Second, the metamorphic transformations in MT-MPC need to be separately implemented for each MPC DSL, which is very costly. BabelFuzz mitigates this cost by using an expressive and configurable intermediate representation. Third, the oracle implementation in DT mode provides a stronger and more efficient oracle than metamorphic transformations, which however are useful in testing specific language features, e.g., ones that cannot be easily implemented in the oracle implementation.

Differential testing. Differential testing [39] is typically combined with program generation (e.g., [19, 34, 51, 56]) to find logic bugs in compilers (e.g., [14, 30, 44]). At a high level, two or more compilers are run on the same input program, and their results are compared. In case of a discrepancy, a bug is detected, but it may be unclear which compiler is to blame. Although this technique has been successfully applied to several mainstream compilers, it is difficult to directly use it to test MPC compilers—each MPC compiler supports its own DSL. To address these issues, BabelFuzz compares each DSL program to the oracle implementation, and as a result, it becomes clear which compiler is buggy.

Differential testing has also been used to test program analyzers (e.g., [19, 21, 24, 27, 43, 48, 53]). In fact, many compilers also perform (typically lightweight) program-analysis passes—such as constant propagation or dead-code elimination—for performance optimization.

Our oracle implementation could also be seen as a form of specification-based testing (e.g., [9, 40, 46]); it is essentially a specification for the correct behavior of each DSL program.

Metamorphic testing. Metamorphic testing [13] is a common technique for testing both compilers (e.g., [17, 28, 29, 45]) and program analyzers (e.g., [22, 23, 36–38, 41, 49, 52, 54, 55]). In some domains, such as Datalog engines [38] or zero-knowledge pipelines [23], each tool supports a different language, and suitable intermediate representations had to be designed, like in BabelFuzz.

7 Conclusion

We presented BabelFuzz, a framework for detecting logic bugs in MPC compilers. By generating programs in an expressive IR and translating them to diverse MPC DSLs and a mainstream programming language, BabelFuzz supports metamorphic testing and enables a strong differential-testing oracle. This design keeps the effort of supporting new compilers low. Our evaluation on four MPC compilers shows that BabelFuzz is effective and efficient, uncovering 27 new logic bugs. In future work, we plan to support additional language features and richer translations.

Data Availability

Our tool is publicly available at https://github.com/Rigorous-Software-Engineering/BabelFuzz and archived at https://doi.org/10.5281/zenodo.19221619.

Acknowledgments

We thank the MPC-compiler developers for their valuable work and assistance. This work was supported by the Austrian Science Fund (FWF) 10.55776/DOC1345324.

References

[1] [n. d.]. Athos. https://github.com/mpc-msri/EzPC/tree/master/Athos.
[2] [n. d.]. The EMP Toolkit. https://github.com/emp-toolkit.
[3] [n. d.]. EzPC: Easy Secure Multiparty Computation. https://github.com/mpc-msri/EzPC.
[4] [n. d.]. MP-SPDZ: Versatile Framework for Multi-Party Computation. https://github.com/data61/MP-SPDZ.
[5] [n. d.]. MT-MPC: Metamorphic Testing of Secure Multi-Party Computation (MPC) Compilers. https://github.com/winnylyc/MT-MPC.
[6] [n. d.]. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. https://github.com/circify/circ/tree/mpc_aws.
[7] [n. d.]. SIRNN: Secure Inference for Recurrent Neural Networks. https://github.com/mpc-msri/EzPC/tree/master/SIRNN.
[8] Fabrice Benhamouda, Shai Halevi, and Tzipora Halevi. 2019. Supporting Private Data on Hyperledger Fabric with Secure Multiparty Computation. *IBM J. Res. Dev.* 63 (2019), 3:1–3:8. Issue 2/3.
[9] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.
[10] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *EuroS&P*. IEEE Computer Society, 496–511.
[11] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. 2023. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. In *SP*. IEEE Computer Society, 848–863.
[12] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.
[13] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.
[14] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *PLDI*. ACM, 85–99.
[15] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *ISSTA*. ACM, 95–105.
[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*. The Internet Society.
[17] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *PACMPL* 1 (2017), 93:1–93:29. Issue OOPSLA.
[18] Xiao Dong, David A. Randolph, Chenkai Weng, Abel N. Kho, Jennie M. Rogers, and Xiao Wang. 2021. Developing High Performance Secure Multi-Party Computation Protocols in Healthcare: A Case Study of Patient Risk Stratification. *AMIA Summits Transl. Sci. Proc.* 2021 (2021), 200–209.
[19] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ISSTA*. ACM, 1219–1231.
[20] Valerie Fetzer, Marcel Keller, Sven Maier, Markus Raiber, Andy Rupp, and Rebecca Schwerdt. 2021. PUBA: Privacy-Preserving User-Data Bookkeeping and Analytics. *Cryptol. ePrint Arch.* (2021), 1683.
[21] Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Constraint-Based Test Oracles for Program Analyzers. In *ASE*. ACM, 344–355.
[22] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *PACMSE* 1 (2024), 1656–1678. Issue FSE.
[23] Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2025. Fuzzing Processing Pipelines for Zero-Knowledge Circuits. In *CCS*. ACM, 783–797.
[24] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.
[25] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS*. ACM, 1575–1590.
[26] Marcel Keller and Ke Sun. 2022. Secure Quantized Training for Deep Learning. In *ICML (PMLR, Vol. 162)*. PMLR, 10912–10938.
[27] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.
[28] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*. ACM, 216–226.
[29] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. ACM, 386–399.
[30] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *ISSTA*. ACM, 327–337.

[31] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *CACM* 52 (2009), 107–115. Issue 7.
[32] Yichen Li, Dongwei Xiao, Zhibo Liu, Qi Pang, and Shuai Wang. 2024. Metamorphic Testing of Secure Multi-Party Computation (MPC) Compilers. *PACMSE* 1 (2024), 1216–1237. Issue FSE.
[33] Xin Liu, Xiaoyu Fan, Rong Ma, Kun Chen, Yi Li, Guosai Wang, and Wei Xu. 2024. Collaborative Fraud Detection on Large Scale Graph Using Secure Multi-Party Computation. In *CIKM*. ACM, 1473–1482.
[34] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *PACMPL* 4 (2020), 196:1–196:25. Issue OOPSLA.
[35] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *ISSTA*. ACM, 248–260.
[36] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *ESEC/FSE*. ACM, 639–650.
[37] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.
[38] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*. ACM, 236–247.
[39] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.
[40] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.* 27 (2017). Issue 6.
[41] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *ICSE*. IEEE Computer Society, 550–562.
[42] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler Infrastructure for Proof Systems, Software Verification, and More. In *SP*. IEEE Computer Society, 2248–2266.
[43] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *PACMPL* 5 (2021), 1–19. Issue OOPSLA.
[44] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *ICSE*. ACM, 203–213.
[45] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*. ACM, 849–863.
[46] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.
[47] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *IJCAI*. ijcai.org, 4873–4882.
[48] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.
[49] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.
[50] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *ICSE*. ACM, 126:1–126:13.
[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, 283–294.
[52] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.
[53] Chengyu Zhang and Zhendong Su. 2024. SMT2Test: From SMT Formulas to Effective Test Cases. *PACMPL* 8 (2024), 222–245. Issue OOPSLA2.
[54] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *ESEC/FSE*. ACM, 237–249.
[55] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *PACMSE* 1 (2024), 722–744. Issue FSE.
[56] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*. ACM, 347–361.
[57] Ian Zhou, Farzad Tofigh, Massimo Piccardi, Mehran Abolhasan, Daniel Robert Franklin, and Justin Lipman. 2024. Secure Multi-Party Computation for Machine Learning: A Survey. *Access* 12 (2024), 53881–53899.

Received 2025-09-08; accepted 2026-03-24