Iron Maiden: Detecting Bugs In The P4 Compiler And Pipeline

Aatish Varma
New York University
akv260@nyu.edu

Peixuan Gao
New York University
pg1540@nyu.edu

***Abstract*—We propose Iron Maiden, a simple test bed architecture to test the P4 compiler in the hope of finding both compiler and semantic bugs. In comparison to common black-box fuzzing, our architecture uses P4's language constructs to create diverse programs. Over a few weeks of testing, we were able to find discrepancies in output between semantically equivalent programs and weirdly worded compiler error messages that we have determined to be bugs in the P4 compiler and BMv2 switch. We have reported two bugs to the P4 organization on Github, one of which has been treated with some enthusiasm by the P4 compiler developers [4]. We hope to build on this work through the use of more formal methods to exploit more vulnerabilities in the compiler to ensure network switches compile programs safely and efficiently.**

1. Introduction

First proposed in 2007, Software Defined Networking (SDN) has changed how network switches operate by allowing a network operator to write a centralized controller to dictate control logic to a cluster of Network Switches. Though originally meant for just the Control Plane, SDN made its way to the Data Plane with the advent of programmable switches, made possible through the P4 programming language. With this new functionality comes the risk of switches incorrectly compiling switch programs leading too packets being routed to the wrong user or the packet data itself being mutated.

Due to the nature of networking, detecting bugs in data plane programming languages like P4 are difficult. Unlike languages like C and Java where a bug can be deduced from simply reading the compiler's output, P4 compiles to a target specific language which is then ingested in to a switch (or a switch simulator like BMv2). The user then must run packets on the switch and read them from an output port to ensure that any changes to them are as intended. Furthermore, any switch functionality is a product of both the logic in the control plane and that logic's implementation in the data plane. As a result, bugs can exist in both planes which makes it even more difficult to ensure that a switch is running as intended.

To mitigate these difficulties, the P4 compiler should be heavily tested so it does not constrain the quality of service network switches were intended to provide. We propose a way to accomplish this goal. By passing programs with varying constructs to the P4 compiler while preserving semantic equivalence between them, the P4 compiler can be efficiently tested for bugs. We elaborate on how to do this in the duration of the paper.

2. Background and Motivation

2.1. Background

P4's compiler is responsible for translating P4 code to a target specific language that runs on a network switch. It is written in C++ and runs multiple target-independent and dependent passes over three stages:

Front-End: Uses 25 distinct passes (40 total passes) to validate, type-check, and perform basic optimizations. Their are approximately 20 distinct transformations done on the original P4 program.

Target Specific Middle-End: P4 provides a library of passes (around 25) that a switch vendor can choose from to implement switch specific policies (e.g. If a vendor wants enums in P4 to be converted to 32 bit integers, then it can specify that policy by using a enum to integer pass provided by P4's library).

Target Specific Back-End: Transforms Intermediate Representation (IR) from the middle-end to machine code run directly on hardware (registers, match-action table). It is responsible for register allocation, instruction selection, and generating a program that is used to pass packets to the switch.

These three stages are in the form of p4c, a P4 compiler front-end and middle-end, and BMv2, a P4 compiler back-end. When we refer to the "P4 Compiler" we are referring to the combination of P4c and BMv2. Bugs in this compiler exist as either Compiler Errors, which will appear at compile time, or non-semantic preserving errors which can only be observed at run time (testing packets) . We hope to exploit both types of bugs in our experimentation.

2.2. Motivation

Our motivation is two-fold:

1. A compiler's soundness is extremely important to allow a developer's program to operate correctly. In Network Switches, this quality is of even more importance, as having correct and on-time data arrive at an end-host is what allows the internet to run smoothly.

2. Long living compilers, even with heavy testing, still have reported bugs more than 10 years after their inception [8]. P416, the latest revision of the P4 language, was proposed in 2016 and it's compiler, P4c, was made public in 2017. Given the age of both P4's language and compiler, we believe that basic testing of the compiler can yield strong results in terms of compiler bugs.

2.3. Related Work

Our work draws on multiple projects which use a variety of techniques to test different compilers. American Fuzzy Lop (AFL) is a generic black-box fuzzer which has been used to test anything from popular applications like Internet Explorer to language compilers like GCC [1]. Though its methodology is both somewhat random and unrefined, it is quite powerful as Sami Liedes proved in 2014 when he was able to detect "34 distinct assertion failures in the first 11 hours" of testing Clang [2].

CSmith [9] is a randomized test generator that was able to find 325 bugs in common C++ compilers like GCC and LLVM, each of which have been around for more than 10 years. The key takeaway of CSmith is it's test generation philosophy - " it generates tests that explore atypical combinations of C language features. Atypical code is not unimportant code, however; it is simply underrepresented in fixed compiler test suites" [9]. This was the basis of our inspiration - Can we marry atypical code with a slight bit of randomness (where we choose to place the atypical code) to find bugs in P4's compiler?

P4Fuzz [5] is another black-box fuzzer used to find bugs in P4. It also uses differential testing meaning that it uses two compilers (BMv2 and EBPF) to compile a generated program and compared the outputs between the two for errors. The authors were able to find 4 bugs with their black-box technique which taught us that P4's compiler has bugs and using a more language focused structure to generate programs could yield even better results.

3. Iron Maiden

We hope to take a few programs and generate multiple semantically and non-semantically equivalent programs from it while exploiting many atypical language constructs in P4. We expand on our methods of doing this below.

3.1. Black-Box Fuzzing

Parser and Lexer bugs are thrown when an input program does not follow the required syntactic structure of the language. These bugs are usually the easiest to find if one can generate multiple programs in the hope that the compiler spits out a compilation error when in fact the program is valid. Vegard [8] took a smart approach in doing this by writing a loose-grammar and applying AFL to that grammar to generate more fine-tuned programs for gcc. Due to time constraints, we chose to take a far simpler approach. Our first course of action was to do generic black-box fuzzing on the P4 compiler in the hope of uncovering basic parser and lexer bugs. We ran AFL on p4c for 4 days on an Amazon EC2 instance. We decided to terminate the run, as no bugs were found.

3.2. Mutation and Mirroring

To exploit many possible P4 language constructs, we defined a set of rules that would either maintain or not maintain semantic equivalence with the original program. We define two different types of rules in the hope of finding both compiler bugs and semantic bugs.

Mirroring Rules : A change to a program that preserves semantic equivalence.

Mutating Rules : A change to a program that does not preserve semantic equivalence.

By mutating a program based on the mirroring and mutating rules we define, we can take a single P4 program and generate multiple similar ones that exploit many P4 language constructs. By covering more constructs, we are more likely to generate a program that is compiled incorrectly.

Our current mirroring rules include:

- Adding semantic preserving bit shifting
- Loading values in to the registers, doing semantic preserving changes, and loading them out
- Dividing actions in to sub actions that are processed consecutively
- Adding Empty Match-Action Tables that are not called
- Replacing if-else statements with ternary operators
- Adding middle variables
- Changing a bit's width
- Adding new types, and changing program variables to that type
- Cancelling computation (e.g. + 1 - 1)

Our current mutating rules include:

- Adding new packet header fields and actions that mutate them
- Adding Non semantic preserving bit shifts
- Using an entirely new program (a last resort mutating rule)

Mutating rules have much significance though they don't preserve semantics. Ultimately, the goal of our experimentation is to generate multiple programs from a single one. Though we won't be able to measure semantic equivalence with a program and its mutated version, our semantic programs use interesting P4 expressions that increase the chance of detecting a compiler bug. We can apply a mirroring rule on a mutated program and test those two programs for semantic equivalence. Thus, mutating rules allow us to have multiple mirroring pairs.

3.3. Example Transformation

Below is an action that defines basic packet forwarding based on IPv4 addresses. Upon entering the switch, the Ingress pipeline's action updates the packet's source and destination addresses updating, and decrements the packets Time to Live (TTL) field.

```
control MyIngress(inout headers hdr, inout metadata meta,
                  inout standard_metadata_t standard_metadata)
{
action ipv4_forward_ori(macAddr_t dstAddr,
egressSpec_t port) {
   standard_metadata.egress_spec = port;
   hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
   hdr.ethernet.dstAddr = dstAddr;
   hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
}
```

By applying multiple mirroring rules from the list above, we generate a new action that utilizes far more P4 language constructs while maintaining semantic equivalence. This new action can be seen below.

```
/* Assume you declare a 64 bit vector field in your
metadata struct (called meta) called temp;  */

control MyIngress(inout headers hdr, inout metadata meta,
                  inout standard_metadata_t standard_metadata)
{

action ipv4_forward_mirrored(macAddr_t dstAddr,
 egressSpec_t port) {
   if ( true ){
      egressSpec_t newPort = port;
      standard_metadata.egress_spec = newPort;
      macAddr_t newDstAddr = dstAddr;
      dr.ethernet.srcAddr = hdr.ethernet.dstAddr;
      meta.temp = hdr.ethernet.srcAddr;
      meta.temp = ((meta.temp & 0xaaaaaaaa)
        >> (true ? 1 : 1)) |
          ((meta.temp & 0x55555555) << 1);
      meta.temp = ((meta.temp & 0xaaaaaaaa) >> 1) |
        ((meta.temp & 0x55555555) << 1);
      hdr.ethernet.srcAddr = meta.temp
      hdr.ethernet.dstAddr = newDstAddr;
      hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
   }
}

}
```

3.4. Test Bed Architecture

Our test bed architecture can be seen in Fig. 2. [Put image of architecture from poster] After mirroring rules are applied to a program, both programs are passed as inputs to the P4 compiler (p4c front-end/mid-end and BMv2 backend). JSON files are produced and are then ingested into different BMV2 switch instances. Then, packets are passed to the switches, which are then processed and sent to the correct output port. We then use tcpdump[3] to inspect packets on the output ports of each switch instance and compare them with each other for semantic equivalence.

3.5. Test Case Generation

We test our P4 Programs by passing different packets to the BMv2 software switch that is running the JSON generated by the P4 Program. To test a given program, the switch should be passed packets that will trigger a variety of match action rules. These rules can only be triggered if the input packets contain header fields that the control plane is choosing to match on.

As a result, only input packets that are expressive enough to trigger match-action rules should be passed to the switch. If not, a risk exists where certain error holding actions may go untested. We formalize this testing process by using P4pktgen [6], an automated test case generator. P4pktgen uses symbolic execution [add a citation] to construct varying packet inputs that can match on all match-actions specified in the control plane. P4pktgen makes it simpler to expose difficult bugs by ensuring that a P4 program is working as intended and that we are testing all possible areas of incorrect functionality.

4. Experiments and Evaluation

4.1. Experiments

4.1.1. Experiment Sets.

4.1.2. Separate Mirroring Program Pairs.

The principle of generating "Separate Mirroring Program Pairs" is concluded in the following figure.

In our experiments, we created our first set of "Separate Mirroring Program Pairs" based on *demo1.p4* and we includes all the specific mirroring rules in the following table.

As can be seen, the original *demo1.p4* program does not includes any control plane logic. In another word, the forwarding logic of *demo1.p4* depends on the control plane

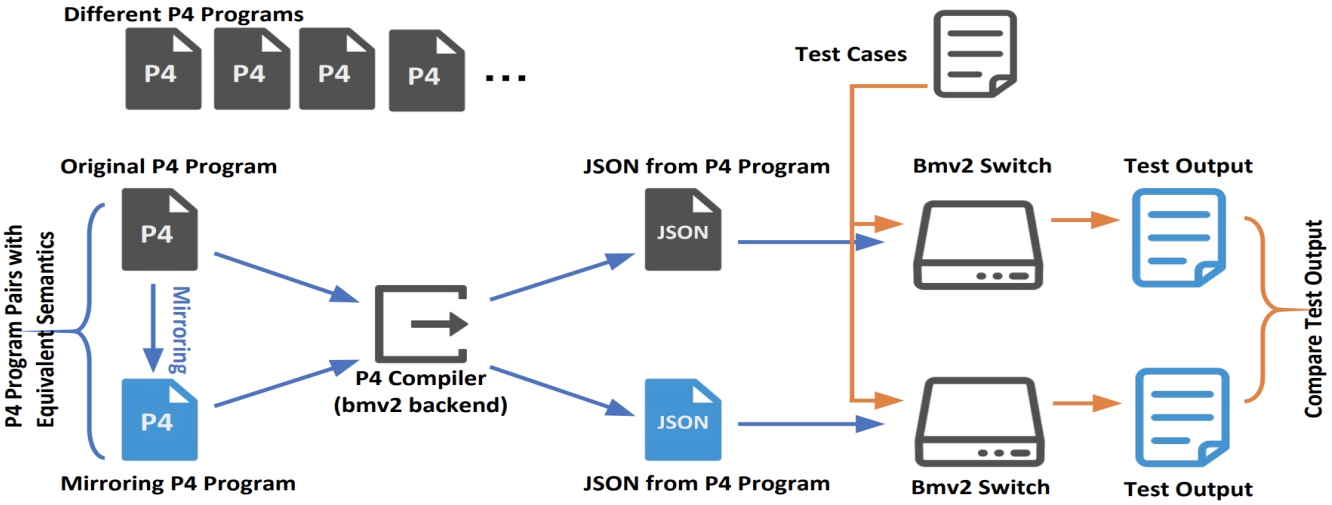

Figure 1. A pipeline of our architecture showing a progression from two P4 programs to their test output

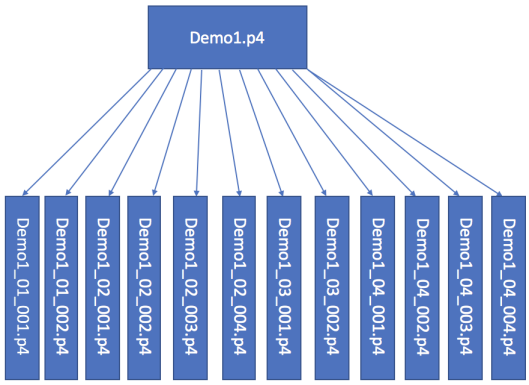

Figure 2. Principle of generating the first set of "Separate Mirroring Program Pairs"

Mirroring Rule	Mirroring Program	Mirroring Operation	Pass the Test Case
Using middle variable for packet header field or metadata	demo1_01_001.p4	The middle variable share the same type with the original header field	Passed
	demo1_01_002.p4	The middle variable is a bit vector instead of the header field's type	Passed
Simple Computations that cancel themselves on the header field and metadata	demo1_02_001.p4	+1 and -1 on dst IP and egress port	Passed
	demo1_02_002.p4	+511 and -511 on egress port (reach the max value)	Passed
	demo1_02_003.p4	-1 and +1 on dst IP and egress port	Passed
	demo1_02_004.p4	-511 and +511 on egress port (reach the min value)	Passed
Adding all match table with NoAction as default rule action (basically do nothing)	demo1_03_001.p4	Adding two such tables	Passed
	demo1_03_002.p4	Adding four such tables	Passed
Adding all match table with default action that cancel themselves	demo1_04_001.p4	Egress port +1 in the first table and -1 in the second one	Passed
	demo1_04_002.p4	Egress port +1 and -1 in both tables	Passed
	demo1_04_003.p4	Egress port +511 in the first table and -511 in the second one (reaching max value)	Passed
	demo1_04_004.p4	Egress port +511 and -511 in both tables	Passed

Figure 3. Principle of generating the first set of "Separate Mirroring Program Pairs"

rules installed into the bmv2 switch at the runtime. In this case, our first set of experiments includes a fixed control plane forwarding rules installed to the original *demo1.p4* and each of its mirroring program.

The control plane rule is rather simple: forwarding all the ipv4 packets with the destination of 10.0.0.0/8 to port 3 and drop other packets. By unifying the control plane rule, all the programs in this set should carry out the same forwarding logic.

4.1.3. Combined Mirroring Program Pairs.

From the previous paragraph, it is easily to notice that the "Separate Mirroring Program Pairs" generate too many mirroring programs at one time, which makes it in-efficient to verify. In this case, we further combined all the mirroring rules together and apply them to the basis program at onetime. Following this principle, each basis program will only generate one mirroring programs that contains all the semantic-preserving translation we have. The validation, in this scenario, only take once to exam the semantic equivalence under all the mirroring rules.

On the other hand, we found by introducing the control plane rule installed during the runtime, the validation process could be slower and the forwarding logic is not only depends on the p4 program but also the installed rules. In this case, we modify the default rules in the p4 programs and hard coded the forwarding logic in the default rules. As a result, there is no need to install any control plane rules during the runtime and the p4 programs could carry out simple forwarding logic we need during the validation process.

By keep the mirroring programs in a low number and adopting the forwarding logic in the default rule, we are able to test more basis program by applying the mutation rules. The following figure describes our principle of creating our second set of "Combined Mirroring Program Pairs".

From the figure, we started with the basis program *basic_demo9.p4* with the similar switch architecture and functions as *demo1.p4*. We hard coded the logic of forward-

Figure 4. Principle of generating the first set of "Combined Mirroring Program Pairs"

ing all the broadcast packets to *port 1* and the other packets to *port 4* in the default rule.

In the mutation process, we first change the matching field of ipv4 *dstIP* to *srcIP*, changing the matching value from broadcast packet (*dstIP = 255.255.255.255*) into *srcIP = 10.0.0.0*. This mutation rule give us a new basis program named *basic_demo10_src_ip.p4*.

In the next step of mutation, we try to play with the packet header fields. We changed the default rules to adding 1 to the *dstIP* if dstIP = 255.255.255.255 as a broadcast packet. After this mutation, the broadcast packets coming out of our switch would have the destination IP address as *dstIP = 0.0.0.0*. We saved this mutated program as *basic_demo11_modifying_dst_ip.p4*.

By applying all the mirroring rules we included in the section 3.2 to all of the three mutated p4 programs, we are able to generate our second set of program pairs including 3 program pairs as figure shows.

4.2. Findings

Testing many semantically equivalent and different programs yielded a few findings and unexpected behavior. They are listed below.

Nested Structs: A struct within a struct is a common feature in languages like C and C++. Because the P4 organization wanted to emulate much of the C language structure in P4, our first plan was to include Nested Structs as a mirroring rule. However, this results in a compiler error - "syntax error, unexpected STRUCT" .This is in no way a bug, but we would like to notify the P4 organization to include in their language specification that this functionality is not permitted.

Typedef: When attempting to do semantic preserving bit shifting on a 2147483648 wide bit-vector, we noticed that the P4 compiler would return a compilation error stating that "Width cannot be negative or zero". This was jarring as our bit-vector width was neither negative nor zero. We referred to the P416 language specification [7] which states that "P4 architectures may impose additional constraints on bit types: for example, they may limit the max size". We then checked the BMv2 specification, which states that "For most of these fields, their is not strict requirement to bit-width"[7]. The only restrictions to bit-width we found in the BMv2 architecture was when a bit vector is used to implement a random or hash function. We realized that bit vector widths are resolved to integer values, which must be positive according to the specification. However, 2147483648 is still positive. We postulated this was a compiler bug and notified the P4 organization on Github. We were written back to by one of the compiler authors saying that P4 resolves bit vector widths to integers, which are expressed as 32 bit vectors. It was not specified in the documentation that integers were expressed as 32 bit vectors, and it is currently being decided between the compiler authors if this should be the case.

BMv2 veths: Input and Output ports on BMv2's software switch are specified using veths (virtual ethernet). Users are also able to specify their own numeric names to veths when instantiating a BMV2 switch. An example is shown below, where 3 port are specified numbered 0 to 2.

```
sudo simple_switch --log-console
    -i 0@veth2 -i 1@veth4 -i 2@veth6 demo5.p4_16.json
```

When we tested our code using a BMv2 switch with 8 ports, we used both even and odd veths unlike above. This was for purely stylistic reasons. However, when using tcpdump, we noticed that packets were arriving at the wrong output port. To determine if this was because of a miscompilation in the P4c, BMv2 backend, or simply an error in the BMv2 switch interface, we reverted back to the use of only even veths. This time, packets were sent to the right output port. We couldn't find any specification in the BMv2 repo mandating the use of only even veths, and we confirmed that our P4 file did not contain any illogic (If there was then the output would have also been incorrect when using even veths). We are deeming this a bug in the BMv2 switch interface, and are reporting it in as an issue on Github.

It is important to note that a difference between outputs of semantically equivalent programs does not necessarily imply a compiler bug. As John Regehr points out [add], languages often allocate a set of language constructs as "undefined behaviors". With undefined behavior, compiler's have leeway in choosing how they process a program. In the case of two semantically equivalent programs that contain undefined behavior, the compiler might choose to return a compiler error in one instance, but choose to exclude the undefined behavior and compile successfully in the other. This can lead to an assessment of a "compiler bug" which is not in fact a "compiler bug".

Unlike languages like C and C++ which have a specified set of language constructs they deem as "undefined", P4 does not have an expansive list of undefined language constructs. P4 only currently defines four areas of undefined behavior:

- out parameters
- uninitialized variables
- accessing invalid headers

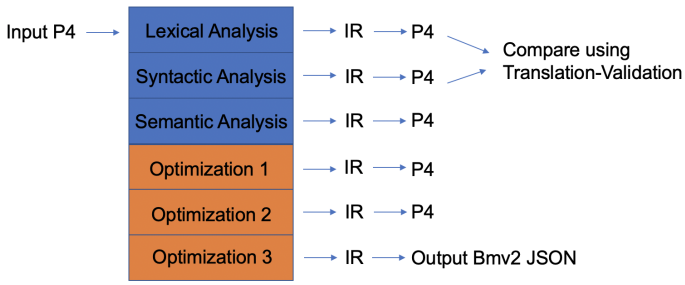

Figure 5. A Translation-Validation approach to verifying the P4 compiler's soundness.

- accessing header stacks with out of bounds instances.

We checked that none of our mirroring or mutating rules fell in this category, which assured that none of our outputs was based on a random choice by the compiler.

5. Future Work

By applying a mirroring rule to a given program, we are generating a semantically equivalent program that uses more language features which may not be handled correctly in a compiler pass. However, the P4 compiler itself is fundamentally a set of mirroring rules applied to the original input P4 program to attain the output. As seen in 2.1, each layer of the compiler produces an immediate representation structured in an Abstract Syntax Tree. By simplifying traversing the tree, one can easily generate a P4 program at each layer of the compiler representing the intermediate changes made from the previous layer.

Instead of producing a P4 program for which we might think their will be a miscompilation, we can directly compare the P4 programs produced at each layer of the compilation with the input program. This is known as Translation-Validation. As Necula [add citation] points out, Translation-Validation's goal is to "check the result of each compilation against the source program and thus to detect and pinpoint compilation errors on-the-fly". This methodology can give us visibility in to the exact optimization and transformations being done at each pass in the compiler. But to verify a compiler layer is performing a semantic-preserving transformation, we would have to guarantee that each packet in put in an input pair (packet passed to original program and the packet beign passed to the current layer's program) is transformed to the same packet as it's counterpart. Luckily, their are "theorem provers" like Microsoft's Z3 [10] which could guarantee this without having to check every input pair. This methodology is far more likely to yield semantic bugs than the methodology utilized in this paper.

6. Conclusion

This paper proposes a test bed that generates multiple semantically equivalent and in-equivalent programs as a way to uncover compiler bugs in the P4 compiler. We are further exploring the use of more formal techniques like Translation Validation to verify the P4 compiler's soundness.

Acknowledgments

We are grateful to Anirudh Sivaraman and Vikas Natesh for their unwavering assistance and patience during this project.

References

[1] American fuzzy loop defination. http://lcamtuf.coredump.cx/afl/.

[2] Fuzzing clang test suite to generate crashing inputs. http://lists.llvm.org/pipermail/llvm-dev/2014-December/079390.html.

[3] Tcp dump. https://www.tcpdump.org/manpages/tcpdump.1.html.

[4] Aatish and Peixuan. P4c bug issue of wrong forwarding port. https://github.com/p4lang/behavioral-model/issues/780.

[5] A. A. Agape and M. C. Dănceanu. P4fuzz: A compiler fuzzer for securing p4 programmable dataplanes. 2018.

[6] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research*, page 5. ACM, 2018.

[7] P4. P4-16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf.

[8] Vegard. American fuzzy loop. http://www.vegardno.net/2018/06/compiler-fuzzing.html.

[9] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[10] Z3Prover. Z3. https://github.com/Z3Prover/z3.