

Blockchain Benchmarking in Practice

The Missing Guide to Diablo

Brandon Stride, Robert* Zhang, Eliséé Djapa

May 11, 2023

Please download this PDF in order to use the hyperlinks throughout.

1 Introduction

One of the critical facets of blockchain technology is its performance and scalability, particularly its transaction processing speed, typically measured in transactions per second (TPS). The TPS rate is a vital performance indicator as it directly impacts the usability and efficiency of a blockchain system. However, blockchain performance is highly dynamic, influenced by various factors like network conditions, consensus mechanisms, and transaction complexity. Therefore, optimizing TPS is a significant challenge in the blockchain field.

Blockchain benchmarking, as done by prior work such as Diablo [Natoli et al., 2021] [Gramoli et al., 2023], is an approach to assess these dynamic performance characteristics. Diablo in particular measures the performance of different blockchain systems using realistic decentralized applications (DApps) as workloads, but is limited by its lack of support for blockchains not implemented in Go and not having smart contracts.

Good benchmarks are essential for developers and users to make informed decisions when choosing the right blockchain for the right scenario. Thus, there is a need for robust yet flexible benchmarking tools that can be adapted to a variety of blockchain systems.

Our paper focuses on extending Diablo’s domain beyond Ethereum-like blockchains implemented in Go, and we use Zcash as a case study. Zcash is a privacy-focused cryptocurrency that uses advanced cryptographic techniques and is based on Bitcoin. We chose Zcash to test the properties of its private-to-private transactions. This report is both a documentation of our work and a reference manual for future Diablo users.

2 Contributions

2.1 Overview


While working towards our benchmarking goals, we ran into many hurdles that do not have clear workarounds. Most of these are a result of vagueness/lack of documentation, incom-


* “Jiaxuan” on course roster

patibility between programming languages, and outdated codebases, all of which severed by the fact that we are working in a complex distributed and networked setting (which none of us is expert in) with multiple moving parts. In particular, we find the most significant issue to be Diablo’s lack of support for blockchains not implemented in Go and not having smart contracts.

Thus, our project has been focused on fixing (or at least remedying) many of these issues and providing thorough guidance for those who may have the same aspirations. This is why we consider this report to be not only a summary of our work up until now, but also a reference manual for those running into the same issues and wishing to get by them effectively.

We hope that through our documentations and tools assisting with using Diablo together with Zcash, all open sourced, it can be easier for the general public to work with and extend on these tools. The following is our two main thrusts towards better tooling for blockchain benchmarking:

- **Comprehensive Diablo Blockchain Integration Guide** We provide a *very* detailed guide on how to integrate a new blockchain into Diablo, covering important concepts, code snippets, and gotchas never mentioned in the official documentation. The authors [Natoli et al., 2021] and [Gramoli et al., 2023] provided only a very high-level overview of Diablo, and both the [website](#) and [repository](#) are relatively void of docs. Our guide fills these gaps, and we also provide a sample specification of Zcash in Diablo, in addition to the provided examples, to serve as a more general template for reference. See 2.2 for some elaboration and 3 for the full guide.
- **cp Tool for SWIG** . To implement a blockchain in Diablo, a user needs to be able to make calls to the blockchain’s code from Go. However, not all blockchains are written in Go or have a Go version. The common tool used by Go users to invoke code written in C++ is [SWIG](#), but it lacks scalability in that it requires all C++ source files to all be in one directory for the tool to work. In large projects like Zcash with many code dependencies, this is simply unsustainable manually. Thus, we fix this by creating a general-purpose tool in OCaml to automate setting up a SWIG project to wrap C++ code in Go. See 2.2 for some elaboration and 4 for the full guide.

We have also led additional efforts beyond the above two directions, which we discuss in 5 and provide links to relevant code/write-ups in 6. Throughout this report, there are many links to our GitHub repository denoted , which contains progress in addition to what is presented here.

2.2 In Detail

2.2.1 Diablo in a nutshell

The Diablo benchmarking application is outlined in [Natoli et al., 2021] and [Gramoli et al., 2023]. The 20,000 foot picture is this: there is a primary machine that coordinates the experiment; it generates the workload, sends it to the secondary machines, launches the benchmark, and aggregates the results. The secondary machines connect to nodes in the

blockchain network and spawn worker threads that mimic individual clients interacting with the blockchain. The workload is a sequence of transactions that request the nodes to run decentralized applications, and these transactions come at a variety of speeds. This way, Diablo can test blockchains in a standardized way, and the exact same workload can be tested on multiple blockchains.

2.2.2 Official Diablo documentation

The fine details of Diablo are completely missing, and the [“Blockchain How-To”](#) page is far from a tutorial. It abstracts the implementation into a tuple, which makes it all even less concrete. The most helpful resources for implementation are a conceptually-commented interface and a few uncommented example implementations.

2.2.3 How we help a Diablo user

We decided to make Diablo more accessible and its documentation more concrete by providing step-by-step instructions for how to implement a blockchain into Diablo. To do this, we implement the blockchain interface almost entirely and mark each “hole” in which blockchain-specific code must go. For each hole, we provide sample implementations and links to remote procedure calls (RPCs) for Solana, Ethereum, and Zcash. We also give a description of what these procedure calls achieve and how they might be found in a blockchain’s source code. This way, a user knows exactly what they need to do to get started.

When writing our Zcash interface for Diablo, we originally wanted to request data from the Zcash RPC via an existing implementation of the RPC client in Go ([zcashrpcclient](#)). However, it has been almost 7 years since the codebase has last been updated, and we soon realized that this implementation is no longer effective as it has become incompatible with one of its dependences ([btcjson](#)) long ago. An instance of the errors we got from `zcashrpcclient` the client is as follows:

```
not enough arguments in call to btcjson.NewImportAddressCmd
    have (string, nil)
    want (string, string, *bool)
```

At this point, we tried cloning `zcashrpcclient` repository locally and specify an older version of `btcjson` that is compatible with how the client is written. However, this proved to be futile, as the furthest we could go is a version in 2018, which is still long after the client code was last updated in 2016. To make matters worse, simply specifying the older version would fail to build the client, as not all historical versions of the packages depended on by the `btcjson` are available.

After surveying `zcashrpcclient`’s repository, we have come to the conclusion that it would be beyond the scope of this project to rectify all the faults (there are a lot!) in its code - it would be an entire project in and out of itself to salvage this 2016 codebase. Thus, we decided to take the alternative approach to instead make our own RPC client, but not from scratch - we port Zcash’s C++ code to Go using the method described in 2.2.4 and later elaborated on in 4.

2.2.4 Working with blockchains in a foreign programming language



Diablo is written in Go and currently exclusively works with blockchains also written in Go, which is a severe limitation. Our approach to extend Diablo to support non-Go blockchains is to take advantage of SWIG, a tool to generate multi-language wrappers for C++ code.

Given C++ and their header files, SWIG wraps the code in such a way that a Go file in the same directory can call functions from the C++ code. If this C++ code is not in exactly the same directory as the Go code, then SWIG breaks down.


We vastly improve the scalability and usability of SWIG in large-scale projects through a tool called `cp` that automatically copies files in nested directories according to a user-provided JSON specification and generates additional batteries to build a complete, working SWIG project. In the ideal case, the user should only need to provide the JSON specification to pinpoint C++ code they want to gather, and `cp` will in most cases do the rest.


3 Comprehensive Diablo Integration Guide

This section will serve as a tutorial to a future Diablo user, and it doubles as a description of our contribution. We will link source code excessively so that a user knows exactly where the code is.

`BlockchainInterface`  “defines all functions that MUST be implemented to integrate a new blockchain.” The comments are 1-3 lines and not fully descriptive of the functions’ responsibilities. Each blockchain implementation must also include `GenericInterface` .

We will walk through each function required in `BlockchainInterface` and describe what precisely it does, provide context code, and explain what RPCs are needed. In instances where code is largely identical for typical blockchains and requires no blockchain-specific calls, we link the code in our GitHub page instead of pasting code snippets. All examples will be with our new Zcash integration, and we explain how to write non-Zcash-specific code where necessary.

Note that the Zcash implementation uses an outdated Golang JSON-RPC client for Zcash . Not all of the functions truly work because of changes to the dependencies, but it gets the idea across. To actually make Zcash work with Diablo, we would have to call the C++ code. For details on this, see 4.

To implement the interface, we make `ZcashInterface` , where each field is thoroughly explained in a comment below:

```

1 type zcashClient = rpc.Client /** Alias for Client type in Zcash RPC package */
2 type txinfo = map[string][]time.Time /** Maps string (hash) to list of times */
3
4 type ZcashInterface struct {
5     /** Primary point of contact for all remote procedure calls (RPCs). */
6     PrimaryConnection      *zcashClient
7     /** All other connections for RPCs in case we need to confirm */
8     SecondaryConnections   []*zcashClient
9     /** SubscribeDone is a channel that reads `true` when it's time to stop
10        ** reading blocks from client. We send `true` here on cleanup. */




```

```



11  SubscribeDone          chan bool
12  /** TransactionInfo maps a transaction to a list of times at which
13      ** it was processed by PrimaryConnection, so we can calculate latency */
14  TransactionInfo        txinfo
15  /** This locks the struct so that we can make updates to the channels
16      ** from within a go routine */
17  bigLock                sync.Mutex
18  /** Is set to `true` when actively processing transactions within
19      ** broadcasted blocks from PrimaryConnection */
20  HandlersStarted        bool
21  /** When the benchmark started */
22  StartTime              time.Time
23  /** Ticks once per second. Upon a tick, we will count the transactions
24      ** that succeeded since the last tick */
25  ThroughputTicker        *time.Ticker
26  /** List of throughput counts for each tick (above) */
27  Throughputs            []float64
28  /** Logs messages for debugging */
29  logger                 *zap.Logger
30  /** Number of failed transactions */
31  Fail                   uint64
32  /** Number of completed transactions */
33  NumTxDone              uint64
34  /** HashChannel is a channel that contains the hash of each new block, so
35      ** we can be notified of a new block (and thus processed transactions).
36      ** This field is Zcash-specific */
37  HashChannel            chan *chainhash.Hash
38  /** Include everything from the linked GenericInterface */
39  GenericInterface
40 }


```

Some of these fields are blockchain-dependent. We discuss here how to choose the types for those fields.

We must have a connection type from which we will make future RPCs. This is called `Client` in Ethereum ; Solana ; and Zcash . Search the RPC folder of your blockchain for a `type Client struct`.

Next, we need to represent a transaction. The transactions can (almost) always be identified by a hash converted to a string. Here are some examples on how to choose a transaction type:

(Zcash) We searched the `Client` functions and found that blocks are instances of `GetBlockVerboseResult`  and its fields included transactions of `TxRawResult` . These have string hashes, so we choose to identify a transaction by a string.

(Ethereum) Go-Ethereum transactions  have hashes that are `atomic.Value`, which by `v.Hash().String()` is a string.

(Solana) The transaction `Transaction` has a `Signature`, and it's convenient to use this type to represent a transaction.

Use these types to fill in the above interface.

The following is a complete description on how to complete the functions in `BlockchainInterface`. The GitHub links at each item point to our Zcash implementation.

- `func (z *ZcashInterface) Init(chainConfig *configs.ChainConfig) Transaction`

See `ChainConfig` and the Zcash `Init()` example. Copy over the nodes from the given `ChainConfig` via `z.Nodes = chainConfig.Nodes`, and make any maps or channels. This is largely blockchain-independent.

- `func (z *ZcashInterface) Cleanup() results.Results Results`

See `Results` for the return type. The `Cleanup()` function stops the ticker, unsubscribes from block notifications, calculates the throughput and latency statistics from the transaction info and throughput lists, and returns the appropriate struct. Nothing is blockchain-specific, so see our Zcash implementation.

- `func (z *ZcashInterface) Start() Transaction`

Like `Cleanup()`, this is not typically blockchain-specific. The purpose of this function is to set the start time of the benchmark and begin a go routine `throughputSeconds()`, which runs a ticker that saves the number of successful transactions each second.

Now we move on to the functions that require some thought and blockchain-specific RPCs.

- `func (z *ZcashInterface) ParseWorkload
(workload workloadgenerators.WorkerThreadWorkload)
([] []interface{}, error) Transaction`

This function parses the workload into blockchain-specific transactions. See the context here:

```

1 func (z *ZcashInterface) ParseWorkload
2     (workload workloadgenerators.WorkerThreadWorkload)
3     ([] []interface{}, error) {
4     z.logger.Debug("ParseWorkload")
5     parsedWorkload := make([] []interface{}, 0)
6     for _, v := range workload {
7         intervalTx := make([]interface{}, 0)
8         for _, txBytes := range v {
9             /** parse the transaction from txBytes which has type []byte */
10            t, err := panic("unimplemented")
11            if err != nil {
12                return nil, err


```

```

13     }
14     intervalTxs = append(intervalTxs, &t)
15 }
16 parsedWorkload = append(parsedWorkload, intervalTxs)
17 }
18 z.TotalTx = len(parsedWorkload) /** save total number of tx */
19 return parsedWorkload, nil
20 }
21

```



Line 10 needs to be implemented. We can assume that the bytes are properly formatted from the workload, so we only need a remote procedure call that converts bytes to a transaction. Here are some examples of such RPCs:

(Zcash) We need a function that parses `var txBytes []byte`, so we search all `Client` functions and find `client.DecodeRawTransaction` , and we fill the hole as follows:

```

var t *btcjson.TxRawResult
t, err := z.PrimaryConnection.DecodeRawTransaction(txBytes)


```

(Ethereum) Declare an empty transaction  and unmarshal the bytes  as follows:

```

t := ethtypes.Transaction{}
err := t.UnmarshalJSON(txBytes)


```

(Solana) Declare an empty transaction  and unmarshal the bytes using the `encoding/json` package as follows:



```

t := solana.Transaction{}
err := json.Unmarshal(txBytes, &t)

```

- `func (z *ZcashInterface) ConnectOne(id int) error` 

Given an index `id` in the node list `z.Nodes`, connect to the desired node. Then, begin listening for blocks from this node, and handle the blocks by parsing them for transactions. Update the transaction info `z.TransactionInfo` accordingly. Most of the implementation is split into two other functions:

1. `func (z *ZcashInterface) EventHandler()` 
2. `func (z *ZcashInterface) parseBlockForTransactions(h *chainhash.Hash)` 

The code context for `ConnectOne` is as follows:

```

1 func (z *ZcashInterface) ConnectOne(id int) error {
2     if id >= len(z.Nodes) {
3         return errors.New("invalid client ID")
4     }
5     /** connect to client here given z.Nodes[id] */




```

```

6      client, err := panic("unimplemented")
7      if err != nil {
8          return err
9      }
10     z.PrimaryConnection = client
11     if !z.HandlersStarted {
12         go z.EventHandler() /** Begin handling events */
13         z.HandlersStarted = true
14     }
15     return nil
16 }

```

Line 6 needs to be implemented (and line 13 is defined later). The client connection can sometimes be quite easy (and sometimes it's not):

(Zcash) We must deal with a few different structs to make a new connection. We make a new `ConnConfig` , and `NotificationsHandler` , and this allows us to subscribe to block notifications upon subscription. Then make the RPC for a new connection .

(Ethereum) Simply dial the client :

```
client, err := ethclient.Dial(fmt.Sprintf("ws://%s", e.Nodes[id]))
```

(Solana) Again, simply dial the client :

```
client := rpc.New(fmt.Sprintf("http://%s", s.Nodes[id]))
```

The event handler will continuously check for new blocks until it reads `true` from `z.SubscribeDone`.

```


1      func (z *ZcashInterface) EventHandler() {
2          z.logger.Debug("EventHandler")
3          /** subscribe to a notifications here */
4          subscriptionChannel := panic("unimplemented")
5          for { /** while true, read from channels */
6              select {
7                  case <- z.SubscribeDone: /** Cleanup called; time to unsubscribe */
8                      /** unsubscribe here; stop getting notifications from channel */
9                      panic("unimplemented")
10                     return
11                 case notification := <- subscriptionChannel:
12                     go z.parseBlockForTransactions(notification)
13             }
14         }
15     }

```


Upon an item in `subscriptionChannel`, a corresponding block is parsed for transactions, and the transaction info is updated. In Zcash, this is done by requesting notifications upon connection. Here are two more examples:

(Ethereum) There is a specific function to subscribe to a client .



```
eventCh := make(chan *ethtypes.Header)
sub, err := client.SubscribeNewHead(context.Background(), eventCh)
```

(Solana) Again, there is a specific function .

```
sub, err := node.RootSubscribe()
```

Now we must parse the blocks that come from the subscription. We use the notifications to retrieve a block and then iterate through its transactions. For each transaction, we update the times at which it has been processed; namely, we say it was processed “now”.

```
1 func (z *ZcashInterface) parseBlockForTransactions(notification) {
2     /** notification is of some type dependent on the blockchain */
3     /** get from z.PrimaryConnection via notification */
4     block, err := panic("unimplemented")
5     if err != nil {
6         z.logger.Warn(err.Error())
7         return
8     }
9     tNow := time.Now()
10    var tAdd uint64
11    z.bigLock.Lock()
12    for _, v := range block./**transactions*/ {
13        tHash := panic("unimplemented") /** optional: get hash from v */
14        if _, ok := z.TransactionInfo[tHash]; ok {
15            z.TransactionInfo[tHash] = append(z.TransactionInfo[tHash], tNow)
16            tAdd++
17        }
18    }
19    z.bigLock.Unlock()
20    atomic.AddUint64(&z.NumTxDone, tAdd)
21    z.logger.Debug("Stats", zap.Uint64("sent",
22                                                atomic.LoadUint64(&z.NumTxSent)),
23                zap.Uint64("done", atomic.LoadUint64(&z.NumTxDone)))
24 }
```

For example, in Zcash, we request a block via a hash with `client.GetBlockVerboseTx(hash)` , where the hash is the notification from the subscription. We then iterate over the transactions , using their hash to identify them.

- `func (z *ZcashInterface) ConnectAll(primaryID int) error` 🔄

In this case, connect to all nodes in the `z.Nodes` field, but call `z.ConnectOne(primaryID)` on only the primary connection. Do not subscribe to the secondary connections, and do not begin more event handlers. No new RPCs are required. See the linked code for more detail.

- `func (z *ZcashInterface) SendRawTransaction(tx interface{}) error` 🔄

Send the given transaction to the primary connection. Count it as a failure or success depending on the result.

```

1 func (z *ZcashInterface) SendRawTransaction(tx interface{}) error {
2     /** Send the transaction and recieve info about it or an error */
3     hash, err := panic("unimplemented")
4     hashstr = hash.String() /** send to string if necessary */
5
6     if err != nil {
7         z.logger.Warn(err.Error())
8         atomic.AddUint64(&z.Fail, 1)
9         atomic.AddUint64(&z.NumTxDone, 1)
10    }
11
12    ParseBlockForTransaction */
13    z.bigLock.Lock()
14    z.TransactionInfo[hashstr] = []time.Time{time.Now()}
15    z.bigLock.Unlock()
16
17    atomic.AddUint64(&z.NumTxSent, 1)
18
19    return nil
20 }
21
```

To implement line 3, cast `tx` to the desired transaction type and send it to the node via RPC. Here are some ways this is done:

(Zcash) First convert to `wire.MsgTx` 🔄 then send to the client with `SendRawTransaction` 🔄. e.g.,

```
hash, err := z.PrimaryConnection.SendRawTransaction(tx.(*wire.MsgTx), true)
```

(Ethereum) Send transaction 🔄 similarly:

```
txSigned := tx.(*ethtypes.Transaction)
err := node.SendTransaction(context.Background(), &txSigned)
```

(Solana) This time it's not as similar, and we must send a transaction with a lot of options 🔄. The finer details are omitted here.

- `func (z *ZcashInterface) Close()` 🐼

Close connections with all clients. This will likely involve a `client.Close()` call as in Ethereum 🐼 and Solana 🐼. In Zcash, it is called `client.Disconnect()` 🐼.

All functions not mentioned above are not actually required as far as we can tell, or they are implemented as part of the generic interface.

We hope that this provides a substantially filled-in interface and direct pointers to RPCs so that a user can easily integrate a new blockchain into Diablo.

4 Tackling Language Incompatibility

As mentioned before, a big issue with Diablo is that it is implemented in Go, and this is a limiting factor in integrating non-Go blockchains like Zcash. While Go is a reasonable choice because quite a few popular blockchains have implementations in it, Diablo could benefit from multi-language support. To bridge this gap, this section will briefly introduce our approach towards improving the usability of SWIG through our `cp` tool.

4.1 SWIG and Go

In a typical SWIG project 🐼, the directory structure would look something like this:

```

/
├── cpp_proj ..... Flat directory containing all header and C++ files
│   ├── hello.cpp
│   ├── hello.h
│   ├── cpp_proj.swigcxx ..... SWIG specification
│   └── cpp_proj.go ..... Go package providing wrapped definitions
├── go.mod ..... Go package consuming wrapped definitions
└── main.go ..... Root of consumer Go package

```

The `cpp_proj.swigcxx` file, in this case, looks like this:

```

1  %module cpp_proj
2  %{
3      #include "hello.h"
4  %}
5
6  %include "hello.h"

```

The `%module` directive tells SWIG to name the wrapper file `cpp_proj`. The content of the `{ %}` pair is copied verbatim into the Go wrapper file generated by SWIG. The `%include` directive behaves largely the same as `#include`, but can also be applied to non-header files. The `cp` tool will generate both to maximize our tool's compatibility with complex projects and lower maintenance overhead.

SWIG support is built into the `go` tool, so to build and run `main.go`, one would simply run the command `go run main.go`. However, once we try to introduce subdirectories like in this case 🐼, we would run into cryptic errors like the following:

```


/opt/homebrew/Cellar/go/1.20.3/libexec/pkg/tool/darwin_arm64/link:
running c++ failed: exit status 1
Undefined symbols for architecture arm64:
  "print_a()", referenced from:
      __wrap_print_a_cppproj_d4c5c25fb00afb76 in 000002.o
  "print_b()", referenced from:
      __wrap_print_b_cppproj_d4c5c25fb00afb76 in 000002.o
ld: symbol(s) not found for architecture arm64
clang: error: linker command failed with exit code 1
(use -v to see invocation)

```

By inspecting the steps taken by `go build`, we determined that Go fails to maintain the original nested hierarchy of the directory before compiling and linking the build artifacts, so the `include` statements in the `.swigcxx` file no longer point to the right path. In fact, the `.h` and `.cpp` files in the nested directories are not copied to Go's temporary build directory at all.

4.2 cp

4.2.1 Overview and Usage

`cp`  is aimed at helping users port a large amount of interdependent source files, C++ or not, that are potentially buried in nested directories, to a different language like Go using SWIG. We have specifically designed it so that all the user needs to provide is a JSON specification file in a very concise and intuitive format, to declare the names of the files they would like to include in their SWIG project. An example of a `cp` spec file is `spec.json` as follows:

```

1  {
2    "source": "cpp_proj",
3    "target": "cpp_proj_wrapped",
4    "swig": true,
5    "worklist": {
6      "dir": "/",
7      "files": [
8        "a.cpp",
9        "#a.h",
10       {
11         "dir": "subdir",
12         "files": [
13           "#c.h",
14           "c.cpp"
15         ]
16       }
17     ]
18   }
19 }

```

The `source` field is the root directory from which to copy files; `target` is the directory to copy all the flattened files into; `swig` is a boolean flag determining whether to generate SWIG-specific files (in particular the `.swigcxx` file). If false, `cp` can be used as a general tool to systematically flatten and dump directories of files.

The bulk of the spec lies in the `worklist` field. Its `dir` field is the directory in the `source` directory to start the recursive lookups from, and `files` is a list of either individual files or nested subdirectories specified by the same two fields, so on and so forth. In particular, `cp` generates a set of `#include` and `%include` statements for each file whose name starts with `#`. In the case of the above example, files `a.h` and `c.h` will be included in the SWIG specification file, but not `a.cpp` and `c.cpp`.

With such a spec file in hand, the user can simply run `cp` as follows to build a working SWIG project:

```
$ ./cp.exe -spec spec.json
```

For this example, `cp` generates the following file hierarchy:

```

/
├── cpp_proj
└── cpp_proj_wrapped
    ├── wrapped
    │   ├── a.cpp
    │   ├── a.h
    │   ├── b.cpp
    │   ├── b.h
    │   ├── wrapped.swigcxx
    │   └── wrapped.go
    ├── go.mod
    └── main.go

```

where `wrapped` is the default name given to the directory housing the Go package that ports C++ code to Go. Notably, the auto-generated `wrapped.swigcxx` file contains:

```



1  %module wrapped
2  %{
3  #include "a.h"
4  #include "b.h"
5  %}
6  %include "b.h"
7  %include "b.h"

```


Now, one would only need to edit `main.go` to add Go code to invoke wrapped C++ functions. Then, by running `go run main.go` in the `cpp_proj_wrapped` directory, one would be able to compile and run it successfully. In rare cases, one may need to edit the `.swigcxx` file to manually include headers not present as a copied file (e.g., `<iostream>`).

For a realistic example and contents of other generated files, please see our codebase for porting actual Zcash RPC source files to Go [🔗](#). Note that one may not be able to actually

`go run` this project if they do not already have all the required non-user-defined C++ library headers available in scope on their system. Also, this is still an ongoing effort to fully port over this portion of Zcash to be 100% functional. Considering that our goal is to primarily provide the infrastructure to guide future users of these tools, we leave it for future work. For the same reason, this directory is currently self-contained and not referenced by our blockchain interface Go file. (Once it is fully done, it should be easy to simply import it from the interface file and tweak a few names of the called functions to run).

We provide pre-built executables for M1 Mac and Ubuntu , but you may need to build the project to generate one that works on your machine. For details on installing OCaml to build/develop `cp`, refer to our README .

4.2.2 Challenges

JSON is originally designed for JavaScript, a dynamically typed programming language. The absence of compile-time typechecking makes it very easy for one to parse JSON constructs like an array with arbitrary elements. For us though, since we decided to implement `cp` in OCaml, a statically typed language, for reasons of performance and security, we had to perform some fancy transformations  on the raw JSON input to mold it into a format acceptable to be properly assigned an OCaml type.

For instance, elements in the `files` field like `"a.cpp"` and `"#a.h"` are transformed into different variants of the same OCaml type, `File "a.cpp"` and `IFile "a.h"` respectively. The same applies to a file that is a subdirectory - it is transformed into `Dir dir`.

This interesting issue prompts us to think about whether data interchange formats like JSON are really suitable for mission-critical tasks. It may not matter as much for blockchain benchmarking, but in domains like distributed computing, we may really need formats that cater better to statically typed programming languages that are generally considered to be more robust and performant.

5 Discussion

In previous sections, we outlined a general approach towards implementing a blockchain interface for Diablo and demonstrated with a sample Zcash interface. However, we unfortunately were unable to download the whole Zcash node and thus could not tap into the RPC. We tried it a few times, but the download process took too long and was taking much toll on our laptops to the extent that we could not do other tasks. Being able to extract blockchain data from the RPC is crucial to actually benchmarking Zcash in Diablo, so we plan to tackle this issue as a next step.

Another direction for future research is to look into ways to integrate shielded transactions into Diablo. This will require very different (maybe not even feasible) workload specifications and deep understanding of the Diablo system that we just couldn't fit into our plans for this semester.




For our `cp` tool, we plan to utilize a C++ parser to automatically extract dependencies from C++ files so the user will not even have to specify these headers in the JSON specification file. This will require some highly non-trivial engineering, but it will make `cp` even more

robust and reliable. A key challenge will be to locate and include non-user-defined C++ library headers (e.g., `boost`), which may require the user to specify a URL for `cp` to download the header files from.

Diablo was a big challenge for us to understand (for reasons detailed in 2.1), and it was an even bigger challenge for us to use in practice. The scope of this project changed from trying to benchmark a blockchain to providing tools and documentation for a repository that really needed it. We feel that our contributions are valuable to the next Diablo user, and we hope that they can benefit from the time we spent on this project.

6 Appendix

This section presents links to our GitHub repository where there are code and write-ups from us not included in this report but are very useful.

- Breakdown and overview of files in the Zcash codebase 
- Our investigation into what Zcash RPC calls are required by Diablo 
- A brief account of a fragment of the issues we ran into and thoughts we had in this project, most of which have been expanded on in this report 

References

- [Co., 2019] Co., E. C. ([2019]). *Zcash Documentation*. Accessed: [May, 2023].
- [Contributors, 2011] Contributors, G. (2011). Zcash. Accessed: [May, 2023].
- [Gramoli et al., 2023] Gramoli, V., Guerraoui, R., Lebedev, A., Natoli, C., and Voron, G. (2023). Diablo: A benchmark suite for blockchains. *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 540–556.
- [Hopwood et al., 2022] Hopwood, D., Bowe, S., Hornby, T., and Wilcox, N. (2022). Zcash protocol specification.
- [Kappos et al., 2018] Kappos, G., Yousaf, H., Maller, M., and Meiklejohn, S. (2018). An empirical analysis of anonymity in zcash. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 463–477, Baltimore, MD. USENIX Association.
- [Natoli et al., 2021] Natoli, C., Benoit, H., Gramoli, V., and Guerraoui, R. (2021). Diablo: A distributed analytical blockchain benchmark framework focusing on real-world workloads.