Just add *WATER*: WebAssembly-based Circumvention Transports

Erik Chi
University of Michigan
ziyunchi@umich.edu

Gaukas Wang
University of Colorado Boulder
Gaukas.Wang@colorado.edu

J. Alex Halderman
University of Michigan
jhalderm@umich.edu

Eric Wustrow
University of Colorado Boulder
ewust@colorado.edu

Jack Wampler
University of Colorado Boulder
Jack.Wampler@colorado.edu

ABSTRACT

As Internet censors rapidly evolve new blocking techniques, circumvention tools must also adapt and roll out new strategies to remain unblocked. But new strategies can be time consuming for circumventors to develop and deploy, and usually an update to one tool often requires significant additional effort to be ported to others. Moreover, distributing the updated application across different platforms poses its own set of challenges.

In this paper, we introduce *WATER* (WebAssembly Transport Executables Runtime), a novel design that enables applications to use a WebAssembly-based application-layer (e.g., TLS) to wrap network connections and provide network transports. Deploying a new circumvention technique with *WATER* only requires distributing the WebAssembly Transport Module(*WATM*) binary and any transport-specific configuration, allowing dynamic transport updates without any change to the application itself. WATMs are also designed to be generic such that different applications using *WATER* can use the same WATM to rapidly deploy successful circumvention techniques to their own users, facilitating rapid interoperability between independent circumvention tools.

KEYWORDS

censorship, circumvention, transport, network, WebAssembly

1 INTRODUCTION

The arms race between censors and circumventors continues to evolve with new tools and tactics emerging from both sides: Censors deploy new mechanisms that block proxies, and in response circumventors develop new techniques that get around the blocking [1, 34, 39, 40].

Because of its dynamic nature, successful circumvention tools must continually develop and deploy new strategies and techniques to get around emerging censorship. For instance, in 2012, Iran blocked several proxies, including Tor, by using an early form of SSL fingerprinting [27]. In response, Tor developed obfsproxy [28], which encrypts all of its traffic including protocol headers in an attempt to evade protocol fingerprinting attacks [20, 24]. While successful in the short term, this was again insufficient as in censors such as China deployed *active probing* attacks to detect early versions of the protocol [14, 37], which prompted circumventors to develop probe-resistant proxies [32, 38]. Censors then found and exploited other side-channels and vulnerabilities to differentiate circumvention traffic [1, 19]. Once these problems were addressed, censors began using other features to detect and block fully-encrypted proxies such as entropy measurements [39], and circumventors responded by using prefixes that fool these measurements [17, 39].

Discovering, implementing, and operationalizing circumvention techniques like these can be burdensome, requiring new code and configurations to be written, packaged, approved for distribution by app stores, and pushed to users. Furthermore, each circumvention tool may need to write and maintain their own version specific to their environment, potentially built using an entirely different programming language, adding to the cost.

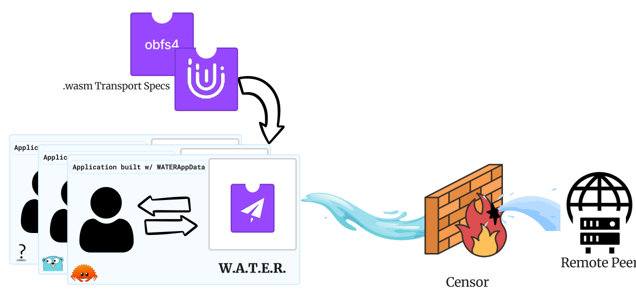

Figure 1: The overview of *WATER*'s role in action. With transport specs defined by `.wasm` files distributed out-of-band, *WATER* can efficiently switch between transports to use.

In this paper, we introduce an approach to ease the burden of developing and deploying new circumvention techniques in the ongoing censorship arms race. Our technique uses WebAssembly, a binary instruction format with runtime support across various platforms, including web browsers and mobile devices. WebAssembly programs can be written in high-level languages such as C or Rust, and compiled into a universal binary that runs on any platform with just a WebAssembly runtime. We extend their use cases with WebAssembly System Interface [4] (WASI) to allow such compiled binaries to make low-level system calls related to network sockets and perform I/O operations by defining an experimental interface for WebAssembly Transport Module (WATM), which will be described in detail in Section 3.2. We also create *WATER*, a runtime library allowing circumvention tools to use portable circumvention techniques from WATMs.

Free and Open Communications on the Internet (1), 22–28

To use *WATER*, a circumvention tool integrates our *WATER* library into their client-side program or application. Then, circumventors can build universal WATMs that implement new circumvention techniques. These WATMs can then be distributed to users over existing data channels, avoiding the need to update the whole app, which involves censored mobile app stores or dealing with blocked CDNs. One use case can be a technique that implements a fully encrypted proxy, written once and distributed to a myriad of circumvention tools as a WATM, despite the tools using different software languages and libraries.

WATER has a distinct advantage over prior approaches that provide similar flexibility, such as Proteus [33] or Pluggable Transports [23]. Where *WATER* leverages WebAssembly, new techniques can be written in one of several (and growing [35]) high-level languages. In contrast, Proteus requires that techniques be written in a bespoke domain-specific language (DSL) which is incompatible with the import of other code or libraries, and thus must be entirely self-contained. Meanwhile, Pluggable Transports must maintain separate, language-specific APIs for each supported programming language (currently Go, Java, and Swift), which are incompatible with each other. In contrast, WATMs can run in any WATER runtime, currently featuring off-the-shelf implementation in Go and Rust with the potential of being implemented with any WebAssembly runtime with WASI support. Furthermore, these WATMs can be compiled from multiple languages, including popular ones such as Rust, Go, Python (with CPython), C (using wasi-libc) [35]. *WATER* is also well-positioned to benefit from future WASI developments as the standard becomes more widely supported and feature-rich.

In the remainder of this paper, we describe our design of *WATER*, implement several proof-of-concept WATMs, and compare their flexibility and performance to existing tools.

2 RELATED WORK

Several prior works have focused on providing *protocol agility* to circumvention tools. We detail these and describe their differences to *WATER* below.

(Tor's) Pluggable Transports [23] offers a standardized interface that different tools can integrate into. This allows circumvention tools (e.g. Tor) that implement the pluggable transport specification to easily add code for new circumvention transports at compile time. However, Pluggable Transport interfaces are language-specific, and currently only Go, Java, and Swift are supported [25], making it difficult to create cross-platform transports that work in multiple projects written in different languages.

Proteus implements an alternate method of dynamically deploying new transports. It compiles text-based protocol specification files (PSF) and executes them at low-level with Rust, improving the flexibility in deployment without losing too much performance [33]. However, Proteus requires the use of a DSL that forces developers to use a Rust-based syntax and adopt a restrictive programming style when developing a PSF. This prevents the direct incorporation of existing tools and increases the difficulty of designing transports from scratch. Also, as Proteus is implemented in Rust, it is challenging to integrate Proteus into projects in other programming languages.

Marionette is a configurable network traffic obfuscation system used to counter censorship based on DPI [13]. It uses text-based message templates to apply format-transforming-encryption (FTE) to encode client traffic into benign looking packets. The templates are constructed using a DSL along with some customizable encoding and encryption operations. The DSL is not Turing-complete and is relatively restrictive. To support more complex protocols, Marionette provides an interface for plugins which requires some degree of recompilation and therefore still requires redeployment.

3 DESIGN

There are two key components in *WATER*: 1) a runtime library to be integrated into a circumvention tool to run WATMs and 2) a WebAssembly Transport Module (WATM) as the hot-swappable `.wasm` binary implementing a particular circumvention strategy or transport encoding technique.

3.1 *WATER* Runtime Library

The runtime library is designed to be easy to integrate into circumvention tools, allowing them to run hot-swappable WATMs that implement different circumvention strategies. The runtime library includes a WebAssembly runtime environment to execute the WATMs and also presents a standard high-level interface for the integrating circumvention client to use any WATM without requiring any knowledge about WebAssembly. This allows the client to make calls to the WATM regardless of the underlying logic and for the WATM to be able to interact with external resources such as network sockets, logging, etc.

As depicted in Figure 2 — when the client calls `_water_dial()` in the runtime library, the WATM is launched, and the WATM-defined *connect* method is invoked. This method may choose to make one or more TCP connections, which it does by calling `dial_host()` *imported* from the runtime library. Then, the runtime library returns a virtual socket to the client. When the client writes to the virtual socket, the runtime library passes the data into a WATM-defined data encoding method, which can transform the data based on the protocol and send it out through corresponding network connections. Similarly, when the client reads from the virtual socket, it does so through the runtime library and WATM-defined data decoding method, allowing the WATM to transform received data from the network. In short, the WATM can make arbitrary transformations to the data sent by the client or received from the network in order to implement any circumvention technique. A step-by-step workflow of dialing a connection is shown in Appendix D.

WATER also supports server-side connections, by similarly implementing corresponding `listen` and `accept`.

3.2 WebAssembly Transport Module (WATM)

A WebAssembly Transport Module (WATM) is a program compiled into a WebAssembly binary that implements a specific set of expected functions that its Host may invoke, allowing the *WATER* runtime to interact in a consistent manner while allowing interchangeable WATMs have the flexibility to apply arbitrary transformations to network traffic. For example, WATMs could wrap a stream in TLS (implementing TLS within the WASM binary), could add reliability layers (e.g. TurboTunnel [15], or could arbitrarily

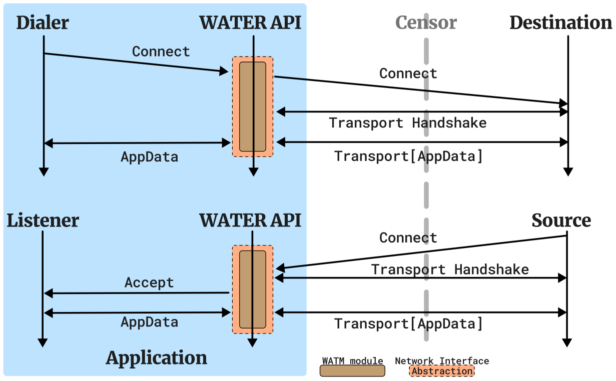

Figure 2: Example connection establishment flows of traditional client (*Dialer*) and server(*Listener*) each using a *WATER* transport. The dialer actively connects to a remote host upon request by caller, with the *WATER* network interface internally managing sockets and IO allowing the WATM to transform the byte stream. Similarly, the listener accept the incoming connections, allowing the WATM to attempt a handshake with the remote host before firing an *accept* hook passing the plaintext end to an upstream handler.

shape traffic by changing the timing or size of packets sent. The set of exposed functions provided by the *WATER* runtime library allows the WATM to interact with the network interface abstraction and the runtime core to manage things like configuration, sockets, cancellation, and logging.

As Rust is one of the most mature languages for writing WATMs, we used it for our initial WATM prototypes. However, we note that any language that can be compiled to WebAssembly could be used in the future, and that the resulting binaries can run in any *WATER* runtime build with WASI-compatible WebAssembly runtime.

3.3 Security Consideration

While WebAssembly provides substantial isolation between the transport module and the runtime library, significantly mitigating the risk of attackers executing malicious code, we note that WebAssembly is not inherently impervious to binary-based attacks [26]. Also, despite the strict interface provided a strong restriction against any arbitrary malicious actions being performed on the host environment, we note that it is still possible for malicious WATMs to make arbitrary connections and potentially leak sensitive data from a circumvention tool loading an arbitrary WATM. As with any software, it is important that we provide a path of trust, using things such as code signing and verification to ensure that the WATM that a client chooses to run is trusted by the deploying application. This is also true of other circumvention tools working with pluggable elements, though those are often integrated at compile time. For example, the Tor project packages and signs pluggable transports that are then launched with `execve` on the client. The WATMs used by *WATER* clients are similar and should be packaged and signed by trusted parties (e.g. circumvention tool developers) before being loaded into a circumvention tool.

4 IMPLEMENTATION

4.1 Runtime Library

We build *WATER* runtime libraries in both Go and Rust to demonstrate the cross-platform and cross-language abilities of our approach, with each providing a native-styled network programming interface for their respective programming language. To avoid excessive duplicated work in implementing a new WebAssembly runtime and keep the design of *WATER* runtime-independent, we only employ standard WebAssembly and WASI interfaces which are available in every standard-complete WebAssembly runtime library [2, 3, 18, 31]. Currently, *WATER* is built with *wasmtime* [2] in Rust and with *wazero* [31] in Go. In addition, we provide starter code, examples, and detailed documentation for developers to build their tool with *WATER*. The open-source repository has been published on GitHub [9–11].

4.2 WebAssembly Transport Module (WATM)

Currently, Rust is the only language with complete official native support for compilation to WebAssembly + WASI, with TinyGo as another very popular choice that allows the compilation of Go code for wasm-wasi. With the support of wasm-wasi being added to more programming languages, we are currently seeing at least 5 languages, including Rust, Go, Python, C, and Zig, becoming possible candidate languages for WATMs.

4.2.1 Provided Examples. A few example WATMs are provided by us to demonstrate the viability.

plain.wasm (available in Rust & TinyGo) implements an *identity* transform WATM that simply copies over the bytes as-received, bi-directionally.

reverse.wasm (available in Rust & TinyGo) reverses the bytes passed (e.g., from `ABCD` to `DCBA`) before writing the result to the other end.

shadowsocks.wasm (available in Rust) demonstrates that WATMs can implement much more complex protocols, such as Shadowsocks. Rather than mimicking Shadowsocks as prior work did, *WATER* is able to build the real Shadowsocks client that works with an unmodified server running `shadowsocks-rust` [7] v1.17.0. To build shadowsocks.wasm, we started with the vanilla shadowsocks-rust and identified 417 lines of code from a file that implements the core part of Shadowsocks (i.e. cryptography and message formatting). We removed all but the default features to minimize the code size and added 142(15% of total) lines as the wrapper code to interface with our WATM specs (more details are provided in Appendix B). Reusing the original codebase allows us to easily apply updates to our shadowsocks.wasm based on upstream changes: we successfully applied the exact patch [21] for shadowsocks-rust defending against the China's blocking of fully-encrypted protocols [39] without any changes to the patch commit.

5 EVALUATION

We evaluated our implementation in Rust, and compared latency and throughput with Proteus [33] and native (Rust) network performance. The evaluation was conducted on the CloudLab testbed [12] on *c6525-25g* (16-core AMD 7302P@3GHz, 128GB ECC RAM).

5.1 Performance Metrics

Travel through	Latency	Throughput
shadowsocks-rust (Baseline)	116us	2310 Mbps
shadowsocks-*WATER*	**+493us**	**2.0% (46.2 Mbps)**
shadowsocks-Proteus	+873us	4.8% (110 Mbps)
Raw TCP (Baseline)	26us	2210 Mbps
plain-*WATER*	**+356us**	**82.8% (1830 Mbps)**
plain-Proteus	+250us	105.4% (2330 Mbps)

Table 1: Latency & Throughput benchmark result comparing to the baseline data, with `msg_size=512`. Worth noting that an implementation may run slow enough to combine multiple messages into one single `send()` and achieve better throughput than baseline due to Nagle's algorithm, with the latter has `TCP_NODELAY` enabled and cannot combine messages.

.

Our analysis of *WATER*'s performance is based on experiments for two different transport protocols comparing *WATER*, Proteus, and native implementations: *plain* and *shadowsocks*. We compared the performance of each on a set of writes using buffer sizes ranging from 1B to 4096B and noticed buffer size can have a noticeable impact on latency and throughput. For baseline native implementations we use raw TCP for *plain*, and vanilla shadowsocks-rust for *shadowsocks*. In Proteus, we use a PSF implementing *the identity transform* for the *plain* protocol, and the shadowsocks PSF implemented from the original paper [33] for the *shadowsocks*.

Focusing on the 512-byte packet size, the *shadowsocks* variant of *WATER* demonstrated throughput comparable to Proteus, but with improved latency. In the *plain* setup, *WATER* closely matched the performance of native TCP in both latency and throughput. Despite WebAssembly's virtualization introducing a discernible overhead compared to native methods, we regard *WATER*'s performance as highly promising. Also, we have noticed it is possible to compile WebAssembly into native machine code with an Ahead-Of-Time (AOT) compiler [31] instead of executing it with an interpreter, which could mitigate such overhead.

The notable degradation in throughput for both *WATER* and Proteus when evaluating shadowsocks is in large part due to the cryptographic operations required. WebAssembly runtimes currently lack hardware acceleration for these operations, resulting in higher latency and lower throughput. However, support for hardware acceleration is being actively considered by the WebAssembly community [36], and we anticipate that this will significantly improve *WATER*'s performance in the future. The performance of cryptographic operations in WebAssembly is examined further in Appendix A.

Lastly, we would mention that the testbed hosts were capable of saturating the 2 Gbps network link with the native transports, which is not a typical speed rate through any real-world Internet access provider. Thus, with shadowsocks-*WATER* achieving a speed rate of around 50 Mbps, we believe it is very unlikely to be the bottleneck in a real-world circumvention scenario. Further performance-based analysis can be found in Appendix C.

6 DISCUSSION

6.1 Advantages and Limitations

Maximized code reuse. Beyond the interchangeability of the WATMs, the use of WebAssembly also enables existing tools (implemented in languages can be compiled to WASM) to easily be converted into new WATMs. Appendix B.2 investigates this further, examining the code changes we made while porting an existing circumvention tool to *WATER*.

WebAssembly Limitations (Temporary). Given the limited official support for WASI in many programming languages, we have only proof-of-concept WATM implemented in Go and Rust available. However, we do see promising trends of programming languages embracing and adopting WASI/WASM compatibility.

We also recognize that the use of WebAssembly introduces non-negligible overhead, due in part to the lack of hardware acceleration support for cryptographic operations. However, WebAssembly is a rapidly evolving technology with a large and active community working to bring features like secure randomness sources, cryptographic acceleration, and network socket access into standards [36]. We expect that these features will only improve the potential of *WATER*.

6.2 Future Work

Our current primary focus is to ensure that *WATER* is a production ready technology. We plan on working with several stakeholders to deploy *WATER* to real world users facilitating rapid and interoperable new circumvention techniques.

WATER could also assist in the discovery of new circumvention techniques. With tools such as OONI [16], Censored Planet [30], and Ripe Atlas [29] the set of probes available is rigid and limited by the software on the available vantage points — promoting a focus on *WHAT* is blocked. Using *WATER* we could instead focus on *HOW* network traffic is blocked by rapidly iterating on probing experiments without redeploying entire applications.

Finally we intend to explore the use of Pseudo Random Functions (PRFs) in WATMs such that the each WATM could have its own unique version of a transport protocol. This would allow for the creation of a large number of unique transports within a class making it more difficult for censors to block users at scale.

We believe that this work only scratches the surface of the potential that WebAssembly has to offer in the circumvention space. We hope that this work inspires further exploration of the use of WebAssembly in circumvention tools.

7 CONCLUSION

In the paper we presented *WATER*, a novel approach using WebAssembly to build circumvention transports that are rapidly deployable by lowering the barriers of deploying circumvention techniques, and simplify the deployment cycle down to the delivery of a new binary file to user's device. We use WebAssembly to provide a sandboxed environment for safely running these binaries, and provide programmable libraries to facilitate integration into existing circumvention tools and/or building new transport modules without knowledge about WebAssembly.

REFERENCES

[1] Alice, Bob, Carol, Jan Beznazwy, and Amir Houmansadr. 2020. How China Detects and Blocks Shadowsocks. In *Proceedings of the ACM Internet Measurement Conference* (Virtual Event, USA) *(IMC '20)*. Association for Computing Machinery, New York, NY, USA, 111–124. https://doi.org/10.1145/3419394.3423644

[2] Bytecode Alliance. 2019. Wasmtime. https://wasmtime.dev/

[3] Wasmer Authors. 2018. Wasmer: Run, Publish & Deploy any code, anywhere. https://wasmer.io/

[4] Lin Clark. 2019. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[5] Frank Denis, sonder joker, Richard Zak, Daiki Ueno, Dan Gohman, Alex Crichton, Stephen Doyle, and Piotr Sikora. 2020. WebAssembly/wasi-crypto: WASI Cryptography API Proposal. https://github.com/WebAssembly/wasi-crypto/blob/main/docs/HighLevelGoals.md. Accessed on 2024-02-11.

[6] ShadowSocks developers. 2023. shadowsocks-crypto. https://github.com/shadowsocks/shadowsocks-crypto.

[7] ShadowSocks developers. 2023. shadowsocks-rust. https://github.com/shadowsocks/shadowsocks-rust.

[8] WATER developers. 2023. ShadowSocks patch for mitigating China GFW blocking of fully encrypted traffic showcase PR in WATER. https://github.com/erikziyunchi/water-rs/pull/14.

[9] WATER developers. 2024. water-rs: WebAssembly Transport Executables Runtime – Resilient & Steadfast. https://github.com/erikziyunchi/water-rs.

[10] WATER developers. 2024. water: WebAssembly Transport Executables Runtime. https://github.com/gaukas/water.

[11] WATER developers. 2024. watm: WebAssembly Transport Modules in Go. https://github.com/gaukas/watm.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, Santa Clara, CA, USA, 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[13] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. 2015. Marionette: A Programmable Network Traffic Obfuscation System. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 367–382. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/dyer

[14] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *Proceedings of the 2015 Internet Measurement Conference* (Tokyo, Japan) *(IMC '15)*. Association for Computing Machinery, New York, NY, USA, 445–458. https://doi.org/10.1145/2815675.2815690

[15] David Fifield. 2020. Turbo Tunnel, a good way to design censorship circumvention protocols. In *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*. USENIX Association. https://www.usenix.org/conference/foci20/presentation/fifield

[16] Arturo Filasto and Jacob Appelbaum. 2012. OONI: open observatory of network interference.. In *FOCI*.

[17] Vinicius Fortuna. 2023. OutlineVPN Wiki: Disguise connections with a prefix to bypass protocol allowlists. https://www.reddit.com/r/outlinevpn/wiki/index/prefixing/y

[18] Cloud Native Computing Foundation. 2021. WasmEdge. https://wasmedge.org/

[19] Sergey Frolov, Jack Wampler, and Eric Wustrow. 2020. Detecting Probe-resistant Proxies. In *Network and Distributed System Security Symposium*. https://api.semanticscholar.org/CorpusID:211263635

[20] Sergey Frolov and Eric Wustrow. 2019. The use of TLS in Censorship Circumvention. *Proceedings 2019 Network and Distributed System Security Symposium*. https://api.semanticscholar.org/CorpusID:142503867

[21] gfw report. 2022. ShadowSocks patch for mitigating China GFW blocking of fully encrypted traffic. https://github.com/gfw-report/shadowsocks-rust/commit/d1cf917deebe1999044ca93965a602223de26be7.

[22] gfw report. 2022. Sharing a modified Shadowsocks as well as our thoughts on the cat-and-mouse game. https://github.com/net4people/bbs/issues/136.

[23] Pluggable Transports Working Group. 2014. Pluggable Transports. https://www.pluggabletransports.info/

[24] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. 2013. The Parrot Is Dead: Observing Unobservable Network Communications. In *2013 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers, San Francisco, CA, 65–79. https://doi.org/10.1109/SP.2013.14

[25] Karl Kathuria, Simone Basso, Jon Camfield, vivivibo, and David Goulet. 2017. Pluggable-transports/pluggable-transports-spec: This is a repository to track issues and suggestions to the Pluggable Transports Spec. https://github.com/Pluggable-Transports/Pluggable-Transports-spec

[26] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Virtual Event, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[27] phobos. 2012. Iran partially blocks encrypted network traffic. https://blog.torproject.org/iran-partially-blocks-encrypted-network-traffic/

[28] The Tor Project. 2012. Tor Project: obfsproxy. https://web.archive.org/web/20120228130444/https://www.torproject.org/projects/obfsproxy

[29] RIPE Ncc Staff. 2015. Ripe atlas: A global internet measurement network. *Internet Protocol Journal* 18, 3 (2015), 2–26.

[30] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. 2020. Censored planet: An internet-wide, longitudinal censorship observatory. In *proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 49–66.

[31] Inc. Tetrate.io. 2020. wazero. https://wazero.io/

[32] The Tor Project . obfs4 Specification. https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc/obfs4-spec.txt.

[33] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. 2023. Proteus: Programmable Protocols for Censorship Circumvention. In *Free and Open Communications on the Internet (FOCI)*. Lausanne, Switzerland, 50–66. Issue 2.

[34] Gaukas Wang, Anonymous, Jackson Sippe, Hai Chi, and Eric Wustrow. 2023. Chasing Shadows: A security analysis of the ShadowTLS proxy. In *Free and Open Communications on the Internet*. 8–13. Issue 1. https://www.petsymposium.org/foci/2023/foci-2023-0002.pdf

[35] WebAssembly.org. 2023. Compile a WebAssembly module from.... https://webassembly.org/getting-started/developers-guide/

[36] WebAssembly.org. 2023. WASI Proposals. https://github.com/WebAssembly/WASI/blob/main/Proposals.md

[37] Philipp Winter. 2013. GFW actively probes Obfs2 Bridges. https://gitlab.torproject.org/legacy/trac/-/issues/8591

[38] Philipp Winter, Tobias Pulls, and Juergen Fuss. 2013. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society* (Berlin, Germany) *(WPES '13)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/2517840.2517856

[39] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. 2023. How the Great Firewall of China Detects and Blocks Fully Encrypted Traffic. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2653–2670. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-mingshi

[40] Diwen Xue, Reethika Ramesh, Arham Jain, Michalis Kallitsis, J. Alex Halderman, Jedidiah R. Crandall, and Roya Ensafi. 2022. OpenVPN is Open to VPN Fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 483–500. https://www.usenix.org/conference/usenixsecurity22/presentation/xue-diwen

A CRYPTOGRAPHY PERFORMANCE IN WEBASSEMBLY

WebAssembly is a new technology still in its early stage of development, and provides an isolated virtual execution environment like other virtual machines (VM). Therefore, it is not too surprising that it inherently lacks the support for native hardware acceleration for cryptographic operations (e.g. SIMD or AESNI). However, it is still important to evaluate and understand the performance of WebAssembly in terms of cryptographic operations. We present results of AES operations within WebAssembly in Table 2.

Configuration	Native	WebAssembly
AES_256_GCM - 256B	230 µs	5300 µs
AES_256_GCM - 1.09G	200 s	500 s
CHACHA20_POLY1305 - 256B	320 µs	5400 µs
CHACHA20_POLY1305 - 1.09G	180 s	200 s

Table 2: Crypto Performance - run on 2021 Macbook Pro M1 Max, 10-core CPU, 64GB RAM. Rounded average for 5 trials.

It is also worth noting that it is possible for an optimizing compiler of WebAssembly to allow efficient cryptographic features, and there is also an existing proposal named `wasi-crypto` that defines a set of WebAssembly-native APIs to import cryptographic operations from host [5].

B IMPLEMENTING SHADOWSOCKS.WASM

We have been developing two distinct versions of `shadowsocks.wasm`: one original and one patched version according to *shadowsocks-rust* to counteract blocking and demonstrate the feasibility of circumventing GFW. Both versions of `shadowsocks.wasm` are designed to handle Shadowsocks' core functionalities, including encryption, decryption, and packaging, within the *WATER* environment.

B.1 PoC version shadowsocks.wasm

The PoC version utilizes the client and server implementation of the *shadowsocks-rust* library, integrating `shadowsocks.wasm` to just run the main logic for the protocol, which is basically packaging code with *shadowsocks-crypto* [6]. The initial challenge we faced was the limitation of WASI being in its developmental phase, currently supporting only 32-bit targets for compilation. However, we found that a 32-bit usize integer suffices for the core functionalities of encryption, decryption, and packet transmission. We plan to continually update our runtime library to follow the advancements in WASI's latest standard.

B.2 Porting from shadowsocks-rust

shadowsocks-rust **(32%)**	*WATER* _SS **(85%)**
168-242	80-154
...	...
2548-2687	810-922

Table 3: Matched lines in the *WATER*-shadowsocks implementation compares to the official shadowsocks-rust

We minimize the required changes to shadowsocks-rust's client code by identifying the protocol specification section (i.e. encryption, decryption, message framing) and reduced the feature support to only AEAD ciphers and direct connections(act like a transparent relay) for now, along with tunnel creation for asynchronous networking. We also had to implement a SOCKS5 listener to directly handle the incoming connections from external web browsers. Table 3 showcases a segment of the comparative analysis between the core logic implementations in *shadowsocks-rust* and *WATER*-SS. The analysis indicates that, of the total 927 lines of code in *WATER*-SS, 785 lines (approximately **85%**) correspond with those in the official *shadowsocks-rust* implementation. The remaining **142 lines** are primarily comprised of glue code, intentionally incorporated to incorporate all previously discussed enhancements. Notably, the amount of glue code should remain approximately consistent and does not proportionally increase with the expansion of the protocol implementation.

B.3 Patching against GFW

The patch we applied was developed in response to China's move last year to block fully encrypted protocols, as reported by [39]. This particular implementation, designed to mitigate the blocking of shadowsocks by the GFW, was proposed by gfw-report [21] and discussed in detail on Net4People [22]. Our implementation of `shadowsocks.wasm` successfully incorporates this patch without necessitating any modification made to the patching commit.

Table 4 showcases the code comparison result of the output of `diff` on the commit changes made while patching the official *shadowsocks-rust* and *WATER*-shadowsocks. The specific commits compared are the gfw-report *shadowsocks-rust* patch commit [21] and the *WATER*-shadowsocks patch commit [8], where it's obviously showing that the changes in *WATER* is matching exactly the same changes in the official *shadowsocks-rust* patch commit ignoring logging.

*shadowsocks-rust*_diff.txt **(98%)**	*WATER*-SS_diff.txt **(99%)**
1-5	1-5
8-196	6-194
198-319	195-316

Table 4: Matched lines from running the `diff` command on patches between official-SS and *WATER*-SS

C LATENCY AND THROUGHPUT

We also present a detailed Table 5 outlining benchmark results for both latency and throughput across varying single packet sizes. In the table, raw TCP serves as the baseline for comparisons in the plain mode as we discussed in Section 4.2.1. Additionally, in Figure 3, we compared the latency and throughput of Shadowsocks implementations using *WATER* and Proteus to provide a clearer picture of *WATER*'s performance and to identify the optimal packet size for balancing latency and throughput with *WATER*.

P Size(B)	Raw TCP (Baseline)	*WATER*	Proteus
1	24us / 6Mbps	+354us / 166.7%	+240us / 183.3%
64	25us / 337Mbps	+358us / 99.1%	+241us / 104.2%
128	25us / 656Mbps	+341us / 101.4%	+241us / 102.3%
256	24us / 1240Mbps	+358us / 102.4%	+242us / 100.0%
512	26us / 2210Mbps	+356us / 82.8%	+250us / 105.4%
768	25us / 3200Mbps	+358us / 62.5%	+250us / 97.8%
1024	26us / 3930Mbps	+359us / 52.4%	+251us / 101.3%
2048	51us / 6390Mbps	+339us / 31.1%	+288us / 88.7%
4096	54us / 9770Mbps	+334us / 19.2%	+292us / 57.8%

Table 5: Plain-Relay latency/throughput - CloudLab topology

C.1 General Use Case performance

We conducted more real-world general use case tests on an Apple MacBook Pro 2021, equipped with a 10-core M1 Max CPU, 64GB of unified memory, and a 32-core GPU. The results are presented in Table 6, showcasing performance metrics obtained using `iperf3` to connect from Michigan to a server in San Francisco.

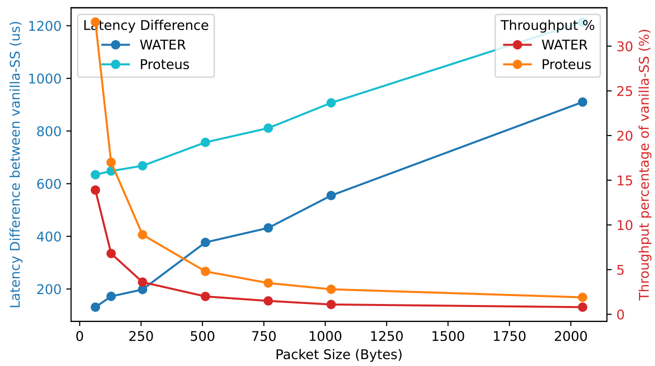

Figure 3: Latency & Throughput Comparison with Vanilla-SS at Different Packet Sizes

Travel through	iperf3 - 10s	iperf3 - 600s
shadowsocks-rust	415 / 411	418 / 418
***WATER*-SS**	**56.5 / 56.5**	**56 / 56**
Proteus-SS	96.6 / 83.5	68 / 67.8

Table 6: Benchmark for General Use Case: Sender / Receiver Throughput (Mb/s) on a MacBook Pro

D WATER WORKFLOW IN DETAIL

In this appendix section, we provide a step-by-step workflow illustration of how *WATER* establishes an outgoing connection in Figure 4.

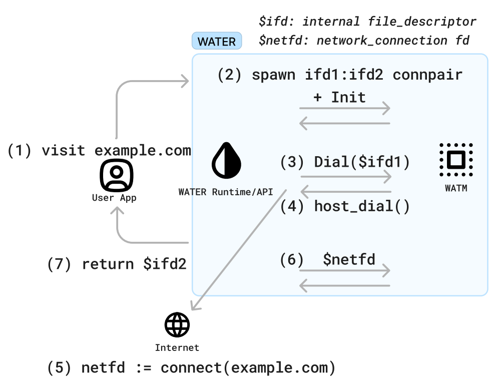

Figure 4: Step-by-step workflow of WATER when a new connection is dialed using WATER API by an integrating application.

E WATM USAGE IN *WATER*

In this section we provide an up-to-date list of APIs we defined for WATM to interact with *WATER* with WASI-based Imported/Exported functions.

E.1 Imported Functions into WATM by *WATER* Runtime

This subsection outlines the functions imported from the *WATER* Runtime to the WATM to allow WATM to interact with host-managed resources with restricted access. Each function is designed to facilitate specific operations within the WATM:

- **`host_dial()`**: A function that initiates a network connection, returning a network file descriptor (`net_fd`).
- **`host_accept()`**: A function designed to accept incoming network connections, similarly returning a network file descriptor (`net_fd`).
- **`pull_config()`**: This function retrieves the configuration settings, issuing a configuration file descriptor (`conf_fd`).

E.2 Exported WATM APIs

This subsection describes WATM API functions exported by each WATM, detailing their purposes and return values to elucidate their roles:

- **`init()`**: Initializes the WATM module, returning an error number (`errno`) as a signed 32-bit integer (`s32`) to indicate success (`0`) or failure.
- **`dial(internal_fd)`**: Used in Dialer role, which establishes a network connection using an internal file descriptor, and returns a network file descriptor (`net_fd`) as `s32`.
- **`accept(internal_fd)`**: Used in Listener role, which accepts an incoming connection on an internal file descriptor, returning a network file descriptor (`net_fd`) as `s32`.
- **`associate()`**: Used in Relay role, which associates an incoming connection with an outgoing connection, typically returning an error number (`errno`) as `s32` to indicate the outcome.
- **`worker()`**: Launches a worker thread and works as the assigned role, returning an error number (`errno`) as `s32`.

E.3 WebAssembly System Interface (WASI)

Besides the above mentioned imported/exported functions, our current WATM spec is based on WebAssembly System Interface 0.1.0 (a.k.a., WASI Preview 1 or `wasi_snapshot_preview1`. A WATM will expect all imports defined by WASI Preview 1 to be made accessible, which SHOULD always be the case out-of-box from any WASI-compliant WebAssembly runtime environment including the ones we mentioned in Section 4.1.