Opt Out at Your Own Expense

Designing Systems for Adversarial Contexts

by

Jack Wampler

B.S., University of California San Diego, 2017

A thesis proposal submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical Computer and Energy Engineering

2023

Committee Members:

Eric Wustrow, Chair

Eric Keller

Tamara Lehman

Shivakant Mishra

Sangtae Ha

Wampler, Jack (Ph.D., Computer Engineering)

Opt Out at Your Own Expense

Designing Systems for Adversarial Contexts

Thesis directed by Prof. Eric Wustrow

Complex systems are multifunctional machines compositionally built of operations in order to complete a set of tasks in alignment with a descriptive protocol. This description is intentionally broad as complex systems are found all around us in life. The functionality of these systems is critical to the continued operation of things like computer processors and communication technologies. However, the actual functionality that complex systems present in implementing their intended protocol almost always goes beyond the expectations of the designers, with the extended functionality often being unavoidable and unremovable.

This work builds up an examination of this **Collateral Centered Design** through case studies that demonstrate the benefits of designing and constructing applications around of the unintended functionalities of a parent system. The first complex system in question is pipelined processors which continue to integrate complex subprocesses for performance optimization. The second is the Internet Protocol (IP) which enables global communication and commerce. The third complex system is one layer up, looking at the protocols spoken over the internet and building a circumvention runtime that allows rapid reconfiguration to match those protocols. For each of these systems I build an empirical understanding of their operational boundaries through structured measurement. I identify goals and adversaries within the context of our desired functionality. Finally I describe in detail each implementation as well as the challenges associated with operating and scaling systems designed in this way.

I hope to demonstrate that leveraging existing behaviors in a complex system to implement new functionality can make avoidance more complicated for adversaries – ideally resulting in an ultimatum where participation in the system requires toleration of the extended functionality.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

This work investigates a design theme applied to systems and exploits in various contexts at various scales. The design principle can be concisely described as such: complex systems often support unintended behaviors beyond their original specification — these unintended behaviors can be composed, and applied at scale to build systems with interesting properties in adversarial contexts.

The behavior of complex systems is rarely binary or easily described; a stark example of the simplicity of a machine capable of arbitrary computation beyond its design is the `mov` instruction which, on its own, is turing complete [105]. This property makes unintended functionality possible, in the case of the `mov` instruction this is the ability to compile a binary to be composed of the single instruction type. We can then imagine the adversarial context where a malware analyst must inspect a program to determine what capabilities the sample supports. If the binary has been mutated using this alternative compiler to consist entirely of `mov` instructions the malware author is able to leverage the complex machine of the single instruction which is simple in construction to significantly complicate the analysts job as the behavior is much less certain in analysis. At the same time, removing the `mov` instruction from the instruction set would be absurd as it is a fundamental building block of modern processors. This concept of applying the unintended behavior of a system in order to achieve a tertiary goal applies to many systems.

This work focuses on the application of these unintended behaviors to specific adversarial situations within the system. This theme is examined in several contexts.

The first context that we consider is processor optimization and malware analysis. Modern processors are deeply pipelined and often optimize for performance at every possible opportunity. In general idle resources are a waste, and one way to prevent idle resources is to optimistically perform common operations based on past execution. This allows processors to take advantage of value locality and speculate on things like conditional branches where execution would otherwise block. However, current speculative execution is not ideal and the state that is impacted by the speculative world is not completely rolled back when (for example) a conditional branch is speculated to be taken but should not have executed based on a bounds check that was blocking. Leaking sensitive information using speculative execution is the focus of the Spectre and meltdown attacks. In contrast, we operationalize speculative execution in order to **Hide** execution. Speculation provides an opaque box in which arbitrary incremental computation can be completed with intermediate states and gadgets automatically obfuscated once the processor terminates speculation. By composing programs of minimal achievable units and intentionally speculating we can obfuscate the true functionality of a binary in seemingly unreachable dead code, with control flow driven by difficult to model speculative predictor state. In the operational context this type of speculative obfuscation is difficult to defend against broadly as speculative units will inevitably miss in real world situations, and removing speculation all-together introduces a significant performance penalty.

From there we take a step back to examine an application of this design theme for good on the internet. Proxies are used to circumvent censorship, and in turn censors attempt to discover and block proxies in order to prevent this circumvention. The ensuing game of cat-and-mouse caused by increasingly stealthy proxy protocols and increasingly vigilant censors is generally constrained by a censors willingness to over-block (break connections for benign traffic) and ability to censor at scale. In order to increase the difficulty along these two parameters refraction networking transitions proxy logic into the middle of the network by partnering with ISPs. This shifts the dynamic of routing and increases the cost of blocking. Refraction as an unintended functionality of internet routing has seen ISP scale deployments for first the TapDance protocol and now its spiritual successor, Conjure. We focus our investigation of the Conjure protocol around the now production scale deployment

and the challenges that arise when a system originally constructed out of unintended functionality in the TCP/IP stack is scaled to millions of users and multiple ISPs.

When considering the effectiveness of collateral centered design in the context of censorship circumvention we can look at more than the IP layer. In the third chapter we address the *WHAT* of protocol censorship. When censors actively monitor traffic for signs of circumvention they are looking for signs that a protocol either stands out, or is not behaving as expected. This allows them to single out perceived illicit traffic and block it. However, there is almost never a fool-proof way to identify a protocol as illicit, and mistakes result in benign traffic being blocked. These **false positives** can have a significant negative impact on online commerce and communication. This provides an incentive for censors to be conservative in their blocking, except in extreme cases. The final chapter explores a pluggable transport design that attempts to lower the barrier to deploying new circumvention protocols. By leveraging the lateral transferability of WebAssembly we implement a proof of concept runtime that could allow transports to be implemented once and deployed to clients of multiple proxies, in multiple languages, on multiple platforms all without rebuilding or redeploying applications. The agility to deploy new circumvention strategies at scale is a significant step towards raising the collateral cost of censorship as circumventors can adapt to blocking faster, transitioning protocols to stay online longer. This forces censors to be more aggressive in the protocols that they block, increasing the collateral cost of censorship.

We find that in practice the collateral centered design principle is effective in attempts to withstand adversarial action, usually at the cost of performance and complexity. In the case of speculative obfuscation, the cost of executing speculative payloads hidden in dead code is a significant performance penalty. In exchange for this performance penalty we are able to hide the true functionality of a binary from all existing reverse engineering and binary analysis techniques. Even in the presence of all current speculative execution mitigations ExSpectre is still able to use the direct branch predictor (BCB) to speculatively obfuscate the true functionality of a binary. In the case of refraction networking, the cost of deploying and maintaining a constellation of refraction stations is in the complexity of the system that handles high volumes of tap traffic

and the connection establishment given the two stage connection process. In exchange for this complexity we are able to provide one of the largest censorship circumvention networks available from the perspective of endpoint IP addresses. Clients currently access around 120k and 10k unique IPv4 and IPv6 addresses respectively every day. Water, the most recent collateral centered design presented in this work, serves to extend the cost for censors along a different axis. Once again the cost to the client is performance as things like cryptographic primitives, that would typically have hardware support, are currently implemented in software lowering steady state performance when using the WebAssembly runtime. In exchange for this performance penalty we are able to dynamically deploy new censorship circumvention strategies. In each of these contexts we emphasize the transition from theory and proposal to novel application and deployment and the benefits and challenges that arise.

1.1 Contextualization

An aspect of all research that often gets omitted is the context in which the work originated. In this brief section I introduce myself as an author and researcher, and provide some context about the events that have motivated several of the underlying themes of this work.

About the Author — As an academic researcher educated in the United States, I have been fortunate in my ability to pursue research without fear of censorship or retribution. A majority of my education has taken place in the Western and South-Western portions of the United States, and I would like to acknowledge the indigenous peoples of these places that have been so influential to me throughout that process – the Navajo, Ute, Arapaho, and Tiwa nations. I have been fortunate to have supportive and talented collaborators and mentors who have played a significant role in shaping my research and career. In addition, as a caucasian, male identifying, American citizen I have been afforded many privileges that have allowed me to freely choose my research topics and pursue them removed from any form of classification. I am grateful for these opportunities and hope to use my position in support of free and open communication on the internet.

While not a complete account, the following are some of the major events impacting the

shape and accessibility of the internet for various peoples in the time since I began my graduate program. These events serve as a backdrop, in a way demonstrating the importance of the design principle that I explore in the work that follows.

Murder of Jamal Khashoggi In October 2018 a US-based journalist and critic of Saudi Arabia's government Jamal Khashoggi was murdered in the Saudi embassy in Istanbul. This politically motivated death was alleged to have come from high ranking officials, and had a stifling message to journalists critical of the Royal Family. *Hong Kong protests* Centering around legislation amending a "Mutual Legal Assistance in Criminal Matters" bill expanding extradition at the governments discretion to countries including mainland China, citizens of Hong Kong took to the streets en masse. Escalation around alleged police brutality incidents in response violent protest behavior extended the original movement for almost two years. Throughout this time the Chinese government actively censored any content covering the protests. *2020 Coronavirus pandemic* An epidemic starting in late 2019 grew into a global pandemic by early 2020. With early uncertainty about the nature of the virus and the potential timelines for a vaccine, isolation became the primary means of prevention. This lead to a significant shift in the way that people interact and collaborate. The internet became the primary means of communication driving a boom in video conferencing and remote work technologies. In some countries the isolation based response was mandated as lock-downs and curfews. Several countries actively censored information about the severity of the pandemic and criticism around the government response. *Turkmenistan protests* A rare instance of public protest in authoritarian Turkmenistan occurred in may of 2020 after significant natural destruction and a denial centered response to the COVID-19 pandemic. Discontent over the lack of government response or support lead to anti-government rallies. The government was quick responded to with arrests and large scale internet censorship. *George Floyd Protests* In late May 2020 George Floyd was murdered by police officers in Minneapolis, Minnesota. This lead to a broad movement of protests across the United States against excessive use of force and qualified immunity for police officers. These protests were met with a significant police response including the alleged use of surveillance technologies like cellular triangulation and IMSI catchers to track and identify organizers. *Myan-*

mar Coup and subsequent protests In February 2021 members of the ruling National League for Democracy (NLD) party in Myanmar were deposed by the commander in chief of the armed forces. Subsequent protests were met with large scale network and social media blackouts. *Cuban protests spring 2021* Economic downturn caused by new sanctions and significantly reduced tourism due to COVID-19 lead to nation wide protests and censorship of the recently rolled out (government controlled) internet service. *Russian invasion of Ukraine* In an escalation of the ongoing conflict between Russia and Ukraine, Russia invaded Ukraine in February 2022 after annexing Crimea in 2014. Along with this came significant censorship of the internet and social media both in occupied Ukraine and Russia itself. *2021 Iranian Presidential Elections* In June 2021 Ebrahim Raisi was elected president of Iran after an election that organizations such as Human Rights Watch called a "sham". As is typical of Iranian elections, the government took stringent measures to prevent the spread of information and communication online leading up to and in the immediate wake of the election. *Russian 2021 Legislative Elections* Rife with controversy, the 2021 Russian legislative elections suffered from accusations of fraud including ballot stuffing and vote manipulation involving a smart voting app. At the same time members of the opposition party were barred or disqualified from running including allies of the opposition leader Alexei Navalny. Pressure from the government media authority, Roskomnadzor, forced policy changes and app removals from both Apple and Google in an attempt to prevent widespread polling. *Kazakh protests 2022* Amidst soaring petroleum prices and income inequality, protests erupted in Kazakhstan in January 2022. As part of a government response that included the deployment of military forces from neighboring countries, the government instituted restrictions on internet access. This culminated in almost a week of rolling nationwide internet blackouts. Further reverberations passed into Turkmenistan where authorities afraid of anti-government action deployed police forces to the streets to enforce a curfew and (reportedly) randomly inspect cellphones. *The death of Mahsa Amini* In September 2022, 22 year old Mahsa Amini was taken into custody by the Iranian morality police on the grounds of wearing an improper hijab. She died under suspicious circumstances while in custody, and the Iranian government routinely refused to share police body camera footage of the incident

or allow access to her body, though images of her injuries leaked online were consistent with a severe beating. This sparked protests as well as counter protests across the country against the continued headscarf laws. After only two days of protests including extensive social media coverage the government instituted rolling nationwide internet blackouts. Protests have continued on and off in the time since. *Chinese Sitong bridge protest* As a political statement during the week before to the 20th National Congress of the CCP an unnamed man staged a protest on the Sitong bridge in Beijing, hanging banners critical of communist party leadership and lighting a fire to draw attention before being arrested. This became a viral event on social media, and was quickly censored by the Chinese government, internet providers, and social media platforms. This has gained the unnamed man notoriety similar to the "Tank Man" of the 1989 Tiananmen Square protests. *20th National Congress of the CCP* Xi Jinping ran for and was elected to an unprecedented and controversial third term as CCP General Secretary in October 2022. This election drew criticism both at home and abroad, though publicized criticism was actively censored within the country. *Israel palestine conflict* In November 2023 after an apparent aggressive attack on Israel by Hamas forces, the government declared a state of emergency and lead a ground based force to take active martial control of the region. As Israel is the direct purveyor of internet access in Palestine and does not have the cleanest history with respect to censorship to begin with, this has lead to widespread internet blackouts for palestinian people attempting to evacuate and avoid the conflict. *Internet Access Legislation* Numerous countries have enacted new legal frameworks in the name of securing against obscenity and illegal activity online. To name only a few so as to provide example — The Russian Sovereign Internet Law passed in 2019 mandates surveillance and the power to partition the country from the global internet. India's IT Rules passed in 2021 shifts liability to hosting providers for content on their platforms, such that companies like Whatsapp and Twitter are monetarily liable for obscenity or any illegal activity as defined by the Indian government. And finally the United Kingdom's 2023 Online Safety Act which mandates that online platforms establish the infrastructure to prevent children from seeing harmful or obscene content as defined by the british government.

Chapter 2

Speculative Execution

Modern processors are extremely complex with multiple highly optimized subsystems working in tandem. One of the primary motivators for processor development is execution speed. In pursuit of this end several subsystems have incorporated predictive mechanisms to leverage statistical similarity in order to opportunistically allocate resources where they would otherwise be underutilized. This allows processors to take advantage of value locality for things like branch prediction and memory prefetching. This strategy of performing work before it is needed based on probabilistic history provides a significant performance gain in the common case because programs do tend to follow low order patterns, provide value locality, and repeat significant portions of control flow. However, in their failure modes this "speculative execution" has extensive and severe unintended functionality.

The Spectre and Meltdown attacks exposed this class of flaws in modern CPU designs, demonstrating that speculative execution allows an attacker to exfiltrate data from sensitive programs. By coercing a processor into perform computation that would otherwise not occur using speculative execution, they show that it is possible to leak the sensitive portions of memory via a side channels to an attacker.

In this chapter we operationalize the failure modes of speculation in a new direction, leveraging speculative execution in order to **hide malware** from both static and dynamic analysis. Using this technique, critical portions of a malicious program's computation can be shielded from view, such that even a debugger following an instruction-level trace of the program cannot tell how its

results were computed. We introduce **ExSpectre**, which compiles arbitrary malicious code into a seemingly-benign payload binary. When a separate trigger program runs on the same machine, it mistrains the CPU's branch predictor, causing the payload program to **speculatively** execute its malicious payload, which communicates speculative results back to the rest of the payload program to change its real-world behavior.

We demonstrate that through structured application of the unintended behavior of highly optimized processor subsystems it is possible to perform arbitrary computation evading all existing reverse engineering and binary analysis techniques. The true functionality of the program is contained in seemingly unreachable dead code, and its control flow driven externally by potentially any other program(s) running at the same time. In the adversarial context of malware analysis, ExSpectre allows malware authors to leverage the transitory nature and opaque behavior of speculation against analysts.

2.1 Introduction

Modern CPU designs use speculative execution to maintain high instruction throughput, with the goal of improving performance. In speculative execution, CPUs execute likely future instructions while they wait for other slower instructions to complete. When the CPU's guess of future instructions is correct, the benefit is faster execution performance. When its guess is wrong, the CPU simply discards the speculated results and continues executing along the true path.

Previously, it was assumed that speculative execution results remain invisible if discarded, as careful CPU design maintains strict separation between speculative results and updates to architectural state. However, recent research has revealed side channels that violate this separation, and researchers have demonstrated ways to exfiltrate results from speculative computation. Most notably, the Spectre vulnerability allows attackers to leak information from purposefully misspeculated branches in a victim process [77]. The Meltdown vulnerability uses speculative results of an unauthorized memory read to sidestep page faults and leak protected memory from the kernel [84]. Both of these vulnerabilities focus on extracting secret data from a process or operating

system. Recent follow-up work has revealed other Spectre "variants", including speculative buffer overflows, speculative store bypass, and using alternative side channels besides the cache [76, 87]. In addition, several attacks have leveraged Spectre to attack Intel's SGX [18, 95, 16], and perform remote leakage attacks [110].

Herein we explore another attack enabled by speculative execution: ExSpectre, which **hides computation** within the "speculative world". Taking advantage of the CPU's speculation to secretly perform computation, we can produce binaries that thwart existing reverse engineering techniques. Because the speculative parts of a program never "truly" execute, we can hide program functionality in the unreachable dead code in a program. Even a full instruction trace, captured by a hardware debugger or software emulator, will be unable to capture the logic performed speculatively. This technique could lead to sophisticated malware that hides its behavior from both static and dynamic analysis.

Existing malware use several techniques to evade detection and make it difficult for analysts to determine payload behavior of reported malware. For example, binary **packers** or **crypters** encode an executable payload as data that must be "unpacked" at runtime, making it difficult to tell statically what a program will do [62]. Malware may also use **triggers** that only run the payload when certain conditions are present, preventing it from executing when it is inside an analysis sandbox or debugger [8, 106].

However, with some effort, these existing malware techniques can be defeated. Analysts can use dynamic execution to unpack malware and reveal its behavior [8], and can use symbolic execution or code coverage fuzzers to determine the inputs or triggers that will reveal malicious behavior [90, 109, 123, 33].

ExSpectre provides a new technique to malware authors, allowing them to hide program functionality in code that appears to not execute at runtime by leveraging Spectre as a feature [61]. This technique defeats existing static and dynamic analysis, making it especially difficult for malware analysts to determine what a binary will do.

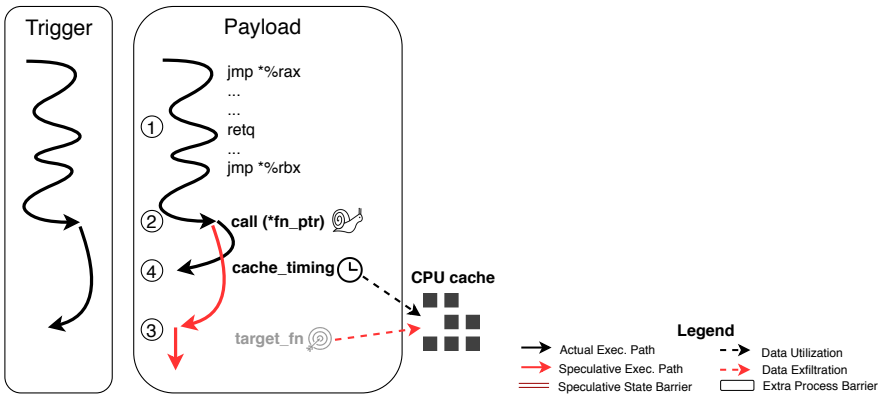

Figure 2.1: **ExSpectre** — Both the trigger and payload binaries perform the same initial series of indirect jumps (Step 1), with tlhe goal of having the trigger program (mis)train the branch predictor. In the payload program, `fn_ptr` has been set to point to the `cache_timing` function but is flushed from cache. Following the pattern in the trigger program, the branch predictor mis-predicts the jump (Step 2) and instead speculatively jumps to `target_fn` (red line). `target_fn` briefly executes speculatively (Step 3), until the `fn_ptr` is resolved and the process redirects computation to the (correct) `cache_timing` function (Step 4). This function then measures information computed in the speculative `target_fn` by measuring a covert cache side channel.

At a high level, ExSpectre consists of two parts: a payload program, and a trigger. The trigger can take the form of a special input (as in typical malware), or an unrelated program running on the same system. When run without the trigger, the payload program executes a series of benign operations, and measures a cache-based side channel[1] . Once the trigger activates—either by the attacker providing specially crafted input, or the trigger program running—it causes the CPU to briefly **speculatively** execute from a new target location inside the payload program.

This target location can be in a region that is neither read nor executed normally by the payload program, making this logic effectively dead code to any static or dynamic reachability analysis. After the CPU discovers the mis-speculation at the target location, it will discard the results and continue executing from the correct destination. However, this still gives the payload program a limited speculative window where it can perform arbitrary computation, and can communicate results back to the "real world" via a side channel. Figure 2.1 shows the variant of ExSpectre that uses a trigger program to mis-train the CPU's indirect branch predictor, causing the payload program to briefly execute a hidden target function speculatively.

It is also possible for a trigger program to be a **benign** program already on the victim's computer. We show this using the OpenSSL library as a benign trigger program in Section 2.5.1, activating a malicious payload program when an adversary repeatedly connects to the infected OpenSSL server using a TLS connection with a specific cipher suite.

We also show it is possible to obviate the trigger program entirely, and instead use **trigger inputs**, which are data inputs the attacker provides directly to the payload, causing the CPU to speculatively execute at the attacker's chosen address. Unlike traditional malware input triggers, these inputs cannot be inferred from the payload binary using static analysis or symbolic execution, as the logic these triggers activate happen speculatively in the CPU, which existing analysis tools do not model. We describe this technique in more detail in Section 2.5.2.

Simulating or modelling the speculative execution path is a difficult task for a program analyst hoping to reverse engineer an ExSpectre binary. First, the analyst must reverse engineer

[1] We note that other side channels could be used in place of a cache

and accurately model the closed-source proprietary components of the target CPU, including the branch predictor, cache hierarchy, out-of-order execution, and hyperthreading, as well as taking into account the operating system's process scheduling algorithm. In contrast, the ExSpectre author only has to use a partial model of these components and produce binaries that take advantage of them, while the analyst's model must be complete to capture all potential ExSpectre variants. Second, the analyst must run all potential trigger programs or inputs through the simulator, including benign programs with real world inputs. Both of these contribute to a time-consuming and expensive endeavor for would-be analysts, giving the attacker a significant advantage.

In order to study the potential of ExSpectre, we implement several example payload programs and trigger variants, and evaluate their performance. We find that a payload program's speculative window is mainly limited by the CPU's reorder buffer, which allows us to execute up to 200 instructions speculatively on modern Intel CPUs. While brief, we show how to perform execution in short steps, communicating intermediate results back to the "real world" part of the payload program. Using this technique, we demonstrate implementing a universal Turing machine (demonstrating arbitrary computation), a custom instruction set architecture that fits within the constraints of speculative execution, and show the ability to perform AES decryption using AES-NI instructions.

Using these building blocks, we demonstrate the practicality of hiding arbitrary computation by implementing a reverse shell in our speculative instruction set, with instructions decrypted in the speculative world. We show that this simple payload is able to perform several system calls in a reasonable time, ultimately launching a dial-back TCP shell in just over 2 milliseconds after the trigger is present.

2.2 Background

Modern CPU designs employ a wide range of tricks in order to maximize performance. In this section, we provide preliminary background as they are relevant to our system, as well as a brief summary of the Spectre vulnerability.

2.2.1 Out-of-Order Execution

Many CPUs attempt to keep the pipeline full by executing instructions **out of order**, with the CPU allowing future instructions to be worked on and executed while it waits for slower or stalled instructions to complete. To maintain correctness and the original (Von Neumann) ordering, instructions are tracked in a **reorder buffer** (ROB), which keeps the order of instructions as they are worked on out of order. Instructions are **retired** from the ROB when they are completed and there are no previous instructions that have yet to retire. Upon retiring, an instruction's results are committed to the architectural state of the CPU. Thus, the ROB ensures that the program (or debugger) view of the CPU state always updates in program execution order, despite out of order execution.

2.2.2 Speculative Execution

CPUs also attempt to keep their pipeline full by predicting the path of execution. For example, a program may contain a branch that depends on a result from a prior slow instruction. Rather than wait for the result, the CPU can **speculatively execute** instructions down one of the paths of a branch, storing the results of the speculative instructions in the ROB. If the guess of the branch target turns out to be correct, the CPU can quickly retire all the instructions it has speculatively executed while waiting. If the guess is incorrect, the CPU must discard the (incorrectly) speculated instructions from the ROB, and continue executing from the correct branch target.

2.2.3 Branch Prediction

When a CPU mispredicts a branch, the speculative execution results are discarded, costing the CPU several cycles as the pipeline is flushed. To minimize this, CPUs employ **branch predictors** that attempt to guess the path of execution. Branch predictors maintain a short history of previous branch targets for a particular branch (e.g. whether a certain branch is frequently taken or not taken), and use this to inform the CPU's guess for speculative execution.

There are two kinds of branches a CPU handles: **direct** and **indirect**. A **direct** branch may either jump to a provided address or continue executing straight through depending on the state of the CPU (e.g. condition registers). While there are only two statically-known targets for a direct branch, the CPU may not know if the branch is taken or not until preceding instructions retire. An **indirect** branch is always taken, but its address is determined by the value of a register or memory address. Direct branches are typically used for control flow such as `if` or `for`/`while` statements, while indirect branches are used for function pointers, class methods, or case statements.

2.2.4 Spectre

In early 2018, researchers revealed the Spectre vulnerability, which allows an attacker to leak information from a victim program [77, 65, 85]. Spectre uses the fact that speculative execution can influence system state via side channel. In Spectre, an attacker mistrains the branch predictor of a CPU running a victim program by providing inputs to it. Once mistrained, the attacker then sends a new input that will cause a different in-order execution path. However, because the CPU's branch predictor has been mistrained, it will still speculatively execute the previous path.

Consider the following code snippet from the Spectre paper [77]:

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

The `if` statement correctly protects an out-of-bounds reads from `array1`. But if the branch predictor makes an incorrect guess on the branch's direction and speculatively executes inside the `if` statement, it may cause a read beyond the boundary of `array1`. The result of this will then (speculatively) be multiplied by 256 and used as an index into `array2`. Although the CPU will not commit the speculative update to `y`, it will still issue a memory read to `array2[array1[x]*256]`, which will be cached. Importantly, even after the CPU realizes the branch misprediction, it does not rollback the state of the cache, as this does not directly influence program correctness. However, the set of cached values is observable to the program via a side-channel: by timing reads to `array2[i]`,

the fastest read will reveal the speculative value of `array1[x]*256`, for any value of `x`. An attacker that is able to perform such a side-channel inference on the cache can learn the speculative result of an out-of-bounds read from `array1`.

Spectre can also be applied to indirect branches. Branch predictors use the history of previous branches to predict the destination of an indirect jump when the destination is not yet known. For direct branches, only one of two destinations (taken or not taken) are possible to speculatively execute. But for indirect branches, a mistrained branch predictor can potentially be coerced into speculatively executing from **any** target instruction in the binary.

We take advantage of the behavior of indirect branch prediction to hide the location of our speculative computation.

2.3 Architecture

ExSpectre malware is comprised of two independent pieces: a payload program, and a trigger. The payload, and some form of the trigger, must be installed on the victim's computer (e.g. via trojan, remote exploit, or phishing). A running payload performs innocuous operations while waiting for the trigger to become present.

One form the trigger can take is another local program that interacts with the payload via the indirect branch predictor. In this case, both programs must run on the same physical CPU. We note that this constraint is not a significant burden, as programs can either use `taskset`, or, if not available, run multiple instances or wait for the OS scheduler to execute both programs on the same core.

At a high level, the trigger program performs a series of indirect jumps in a loop, training the branch predictor to this pattern. Meanwhile, the payload program performs a subset of this jump pattern, then forces the CPU to speculate by stalling the resolution of an indirect branch via a slow memory read. The CPU will (mistakenly) predict the jump to follow the pattern performed by the trigger program, and speculatively execute that destination in the payload program.

The trigger can also take the form of a special input to the payload program, rather than

a separate program. In this case, the payload program parses input data and performs innocuous operations with it. Once the trigger input is provided, it causes the program to **speculatively** overflow a buffer, despite correct bounds checks in the program. The speculative buffer overflow (described in Section 2.5.2) causes the program to speculatively execute at an address chosen by the trigger input and controlled by the attacker.

2.3.1 Threat Model

We assume a scenario where an adversary wishes to hide or obscure the behavior of a malicious program (malware) from an analyst attempting to reverse engineer it. We note this is distinct from the goals of evading malware detection, where malware escapes classification by an anti-virus or other automated tool. While we believe ExSpectre could also be used to make automated detection more difficult, our main focus is on reverse engineer resistance, useful for evading manual classification concerned with malware behavior. For anti-virus evasion, we refer the reader to several existing techniques that are sufficient to defeat existing anti-virus systems [70, 94, 101, 116, 120].

We assume the adversary is able to install binaries on the target machine (e.g. via a trojan or remote exploitation), and the analyst is attempting to determine what the malware will do using traditional debugging tools. We assume the analyst may be aware that speculative execution is used to obfuscate behavior, but does not have special-purpose hardware that allows introspection of the CPU's speculative state. This assumption is realistic, as modern processors do not allow developers or other users to directly interact with proprietary CPU optimizations and features.

We further assume that the malware has a specific trigger that the analyst is not privy to, and the adversary can influence. In our examples, this trigger is often behavior exhibited by some other (potentially benign) process running on the same system as the malware. As the adversary is able to control when such a trigger is deployed (potentially remotely), the analyst will not be able to observe or force this trigger to happen at will. We emphasize that while this may also be true for existing trigger-based malware, analysts can often reverse engineer the trigger out of the malware, for example by observing control flow within the malware and using adaptive fuzzers [142, 114] to

generate inputs that explore other execution paths of the binary. In contrast, ExSpectre malware's trigger influences behavior of the payload program speculatively, making it effectively invisible to the analyst. As with typical malware, the analyst may attempt to reverse engineer the trigger to reveal the malware's behavior, but we will show (in Section 2.4.2.1) how this type of analysis can be defeated.

2.3.2 Indirect jumps

In this subsection, we will describe the trigger program variant, and defer discussion on how input data can be used as a speculative trigger to Section 2.5.2.

In ExSpectre, we cause the CPU to mis-speculate the destination of an indirect branch in the payload program, causing it to speculatively execute instructions that are never truly executed. We term the destination where speculation begins the **speculative entry point**. ExSpectre uses indirect jumps to allow speculative execution from **any** instruction in the payload process' address space. Because it can jump to any instruction, the malware analyst has a difficult task in determining where a payload program's speculative entry point is.

In fact, the location of this entry point is not determined by the payload program, but rather the corresponding trigger program. This means that with only the payload program, an analyst does not posses enough information to find the speculative entry point.

Indirect branch predictors allow the CPU to predict the destination address of a branch based solely off its source address and a brief history of previous branch sources and destinations. While the inner-working details of modern CPU branch predictors are proprietary, it is possible to reverse engineer parts of their behavior, which we do for ExSpectre.

We observe that Intel CPUs consider three types of x86_64 indirect branches: `retq`, `callq *%rax`, and `jmpq *%rax`[2] . We created a simple trigger program that performs a series of indirect branches using `jmpq *%rax` instructions. Between each jump, we incremented `%rax` accordingly to continue on to the next jump. After these jumps, we load a function pointer into `%rax` and do a final indirect

[2] other general purpose registers besides `%rax` can be used as well

Processor	Released	Micro-arch.	Nested Spec.	Indirect jumps	μ-ops (nop)
Intel Xeon CPU E3-1276 v3	2014	Haswell		26	178
Intel Core i5-7200U	2016	Skylake	✓	26	220
Intel Xeon CPU E3-1270 v6	2017	Skylake	✓	28	220
AMD EPYC 7251	2017	Ryzen		4	178

Table 2.1: **Processor features**— We analyzed the capability of ExSpectre on three Intel processors and one AMD. Both Skylake processors were capable of nested speculation (Section 2.3.3.5). Indirect jumps is the number of common training indirect jumps needed in the trigger program to reliably ($> 95\%$ of the time) coerce the payload program to follow the pattern and jump to the speculative entry point specified in the trigger program. μ-ops is the upper bound of μ-ops that can be performed speculatively.

branch using `callq *%rax`. In our trigger, we perform these jumps repeatedly in a loop.

In our payload program, we first perform the same indirect jumps. We ensure the source and destination addresses of these jumps is the same as in the trigger program by manually defining their containing function at a fixed address inside a linker script. We also do the final indirect call to a function pointer, but with two differences. First, the destination in the function pointer is a different address, and second, the memory location of the function pointer itself is uncached. This forces the CPU to predict the destination of the final indirect call while it waits for the function pointer to load from memory. Due to the similar history of branches with the trigger program, the CPU will (incorrectly) predict the destination to be the same as the one in the trigger program, which determines the speculative entry point for the payload. Even though the in-order execution of payload program never executes or even reads from this address, the CPU will briefly execute instructions there speculatively.

In Table 2.1 we analyze the number of necessary training indirect jumps various processors require to consistently ($> 95\%$) have the payload program enter the speculative world at the chosen speculative entry point in the trigger program. We found that 28 indirect jumps was sufficient for our trigger program on each of the test processors to reliably ensure the speculative execution began at the correct speculative entry point.

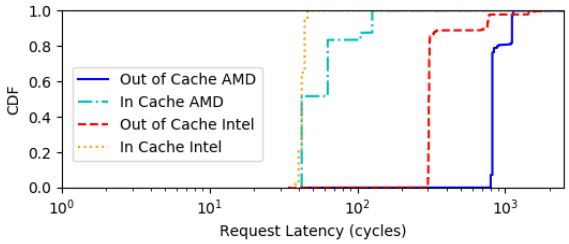

Figure 2.2: **Cache latency** — Cumulative distribution function of the cache hit and miss latency for an Intel Xeon-1270 and AMD Epyc 7251. If a cache miss is used to force CPU speculation, the CPU must wait at least 300-800 cycles before the speculated branch can be resolved. However, we find the CPU is occasionally limited to far fewer instructions speculatively, suggesting another limit is at play.

Eventually, the de-reference of the uncached function pointer in the payload program will be resolved, and the CPU will recognize it has incorrectly predicted the destination of its `callq` instruction. The results from the speculative entry point instructions will be discarded, and the CPU will continue executing from the correct destination. However, the speculative code can change what is loaded into the cache based on its computation, allowing it to covertly communicate its results to the "real world" program.

2.3.3 Limits of Speculative Execution

We performed several experiments to determine how much computation can be performed speculatively, as well as what components are responsible for the limit. We report results from our experiments on an Intel Xeon-1270 (Sandy Bridge), though we note we found similar results across other Intel processor generations, including an i5-7200U (Kaby Lake), an i5-4300U (mobile Haswell), an i5-4590 (desktop Haswell), as well as an AMD EPYC 7251.

2.3.3.1 Cache Miss Duration

When executing instructions speculatively we rely on a memory load of a function pointer from uncached memory. Thus, one limit on our computation comes in the form of the time it takes for the memory read to return with a result (and for the CPU to determine the result was mis-

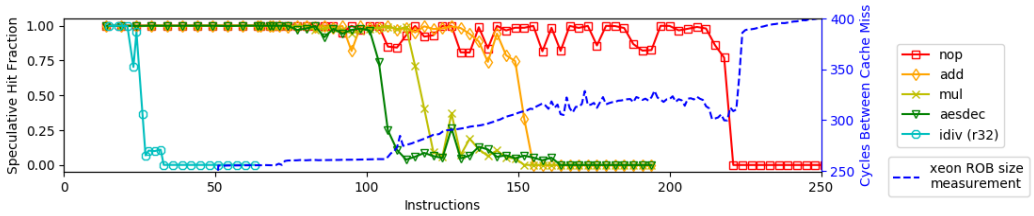

Figure 2.3: **Speculative limits** — We placed a memory read after an increasing number of (speculatively executed) instructions and measured the fraction of times the loaded value was subsequently in cache. This tells us the upper bound of instructions we can reliably execute speculatively. We identify two limitations on the speculative lifetime: cache miss latency resolving the speculative branch, and the CPU's reorder buffer (ROB) size. We observe that different instructions have varying speculative limits: for example, a 32-bit `idiv` can complete only 18 instructions, as each instruction inserts 10 μ-ops into the ROB, while cheaper instructions that use fewer μ-ops can execute more instructions.

predicted). We measured the number of cycles a cache miss takes to return by artificially evicting an item from cache and timing reads from its address. Figure 2.2 shows the CDF of cycles taken. In the typical case, an evicted item takes approximately 300 clock cycles to load from the Level 3 cache (L3), which would allow a limit of roughly 300 speculative instructions (depending on specific cycles per instruction (CPI)) to be executed during that time. We note that when an item is not in L3, it takes considerably longer to load, in theory allowing for thousands of speculative instructions in a significant fraction of runs.

2.3.3.2 Reorder Buffer Capacity

We also measured the capacity of the reorder buffer (ROB) using a method outlined by [133]. We measure the maximum number of cycles taken to perform two uncached memory reads, and vary the number of filler instructions between them. If the number of filler instructions is small, both memory reads will fit inside the ROB, and it can issue their memory reads in parallel. However, if the filler instructions fill the ROB, the second memory read will have to wait for the first to return before it can be issued, causing a noticeable step increase in the cycle count. Figure 2.3 shows this step occurs at approximately 220 instructions for our processor, suggesting a hard upper bound regardless of how long the cache miss takes to resolve.

2.3.3.3 Speculative Instruction Capacity

To verify the upper limit of speculative instructions, we instrumented our trigger and payload programs to test a simple gadget of variable-length before it communicated a signal to the real world via a cache side channel. If the cache side channel revealed no signal in the real world, then we know the speculative execution did not make it to the signal instructions before the mis-speculated branch was resolved.

We also tested whether instruction complexity or data dependencies impact the number of instructions that can be completed. We find that data dependencies and instruction complexity both have an impact on the number of instructions that can be executed. Instruction complexity is determined by the number of μ-ops that the instruction uses, which appears to be what is tracked in the ROB. For instance, on our Skylake architecture, the 64-bit `idiv` instruction takes 57 μ-ops, and we can execute up to 3 of them in the speculative world. Meanwhile, we can execute up to 18 32-bit `idiv` instructions, which each take 10 μ-ops [46]. This suggest we can execute on the order of 175 μ-ops before the speculative world expires.

Most notably instructions that use the extended x86 registers are still valid within the speculative context. Specifically, Intel's hardware accelerated AES-NI encryption and decryption instructions, which each use 128-bit registers. As shown in Figure 2.3, speculative environments can complete a significant number of AES rounds—over 100 rounds in our experiments, more than enough to decrypt a full block using simple AES modes (e.g. AES-CTR). We investigate the use of AES instructions in the speculative environment further in Sections 2.4.2.

We find that when executing speculatively, the number of instructions completed has a soft limit and a hard limit. The duration in cycles applies a soft limit, as shown in Figure 2.3 with the `idiv (32-bit)`, `mul`, and `aesdec` instructions. As we attempt to execute more instructions speculatively, we see a steep drop in in the fraction of trials that are able to signal via the cache-side channel. However, this speculative hit fraction does not drop to zero until the later hard limit, imposed by the number of CPU micro-operations (μ-ops) composing those instructions. Figure 2.3

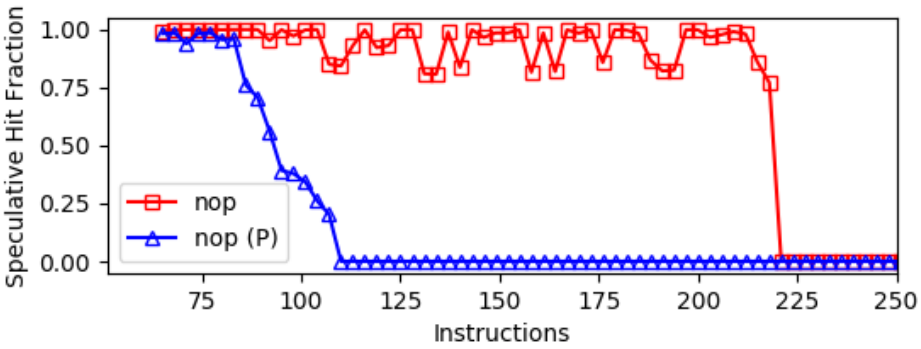

Figure 2.4: **Hyperthreading** — We measured the impact hyperthreading has on speculative execution. Trigger and payload programs running on the same logical core require a context switch to alternate processes, but allows each to have full utilization of the ROB and execution units when they run. Running the programs on parity hyperthreads (denoted by (P)) allows them to run simultaneously without context switching, but we observe this configuration effectively halves the amount that each program can speculatively execute, suggesting that hyperthreads share parts of the ROB or execution units.

demonstrates that the number of μ-ops of the instructions is the major limiting factor that define an upper bound of approximately 150 instructions[3] .

2.3.3.4 Hyperthreading

When running our tests, we assign the payload and trigger program to the same core using `taskset`. We note in the absence of `taskset`, we can run multiple instances of trigger programs to occupy all cores, eventually having the payload program and trigger program become co-resident.

We also explore using hyperthreading, where the CPU presents two virtual cores for each physical core, allowing the OS to schedule programs to each simultaneously. In effect, this can cause the interleaving of instructions between two programs to be much finer-grained: at the instruction level rather than changing only at the OS-controlled context switch. We find that this has two effects on speculative programs. First, the finer-grained interleaving allows for a higher hit rate from the cache, suggesting that each indirect jump pattern is more likely to result in speculatively executing from the intended position. Second, because the physical CPU is being

[3] While `nop` is able to execute up to the full 220 ROB capacity, instructions that do useful work (and/or use multiple μ-ops) cannot reach this limit. In addition, data dependency and execution unit availability add further complications to modelling the exact number of instructions that can be executed speculatively.

shared, it effectively halves the number of instructions that can be run in the speculative context. Figure 2.4 shows the instructions that can be run when running trigger and payload on a single core vs. a pair of hyperthreaded cores.

We note that Single Thread Indirect Branch Predictors (STIBP) have been implemented in most environments to prevent cross thread branch predictor interference. While this does remove the ExSpectre trigger's influence in scenarios where a process runs on an isolated cpu, in most modern environments tasks are scheduled to all available processors. This means that the trigger and payload will be coresident eventually, allowing progress to continue.

2.3.3.5 Nested Speculation

We explore the ability for the CPU to "double speculate", where a second stalled indirect jump while the CPU is already speculating causes it to predict the target and speculate a second time. For instance, suppose a payload program truly jumps to target A, but the CPU is mistrained by a trigger program that jumps to B, thus causing the payload program to speculatively execute at B. At B, suppose there is a second indirect jump, perhaps using the same register as the first jump (which has still not resolved). If the trigger program jumps to C, the payload program may speculate a second time and continue speculative execution at C. Figure 2.6 demonstrates Nested Speculation in action.

We find that not all Intel CPUs support nested speculation. For example, it appears Haswell chips do not speculate while already speculating, but nonetheless support non-nested ExSpectre. Both Sandy Bridge (which preceded Haswell) and Kaby Lake (which followed Haswell) support nested speculation. We find that when a CPU does support nested speculation, there appears to be no limit to nested depth besides the speculative instruction limit. We use a 16-deep nested speculation in Section 2.6.2 to protect speculative decryption keys from reverse engineering.

2.3.4 Speculative Primitive

We summarize our findings into a **speculative primitive**, which allows our payload program to speculatively (and covertly) perform on the order of 100 arbitrary instructions while an accompanying trigger program is running, and communicate a short (e.g. single byte) result to the real world via a cache side channel. These speculative instructions are able to read from any cpu state accessible to the process in the real world including memory and registers, but they cannot perform updates or writes directly. To read memory the **speculative primitive** makes use of the ability to bring things into cache. If a load for an uncached memory location is initiated speculatively it will not finish within the speculative window (meaning no value can be exfiltrated to the real world). However, the memory read is not canceled and the value will be available from cache when the processor accesses it speculatively again. To update memory, the speculative instructions must communicate to the real world. We use a cache side channel to do so, but other side channels compatible with Spectre could also be used [76].

We note a performance tradeoff between the size of communication (e.g. 4 bits vs 8 bits) and the time it takes the real world to recover the result from the side channel. Using Flush+Reload [140] as our cache side channel, recovering the result requires accessing all elements in an array exponential in the size of the result (e.g. 2^8 array reads to recover an 8-bit result). Therefore, there is a performance advantage for keeping the size of the result small, and communicating out small pieces of information that are aggregated by the real world over multiple speculative executions. Meanwhile, smaller channels introduce more overhead in recovering information. We investigate this tradeoff in Section 2.6.2, and find that 8 bits is near optimal in practice.

2.4 Application Payloads

While the amount of computation done in a single speculative execution is small, we demonstrate several applications that can take advantage of multiple speculative runs to carry out computation.

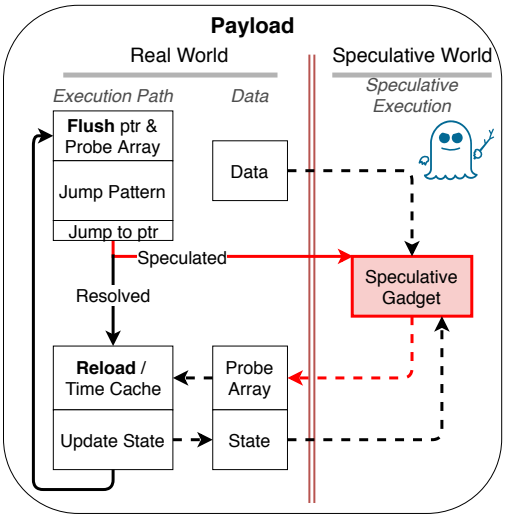

Figure 2.5: **ExSpectre model** — General model of speculative computation within the payload process when triggered. The *Speculative Gadget* has read-only access to all memory within the process, but can only return updates/results via a cache side channel (by accessing the `probe_array`). The process can subsequently *Reload* from the cache side channel to learn the speculatively-computed result, and update the state of the *Real World* process.

As a first step, we observe that the speculative primitive can be used to trivially implement a finite state machine: logic can be done in the speculative world, while updates to the state are communicated to the real world where they are stored. On the next run of the speculative instructions, the state is read from the real world state (along with any inputs), state transitions are computed and communicated back. In this mode, the state is maintained by the real world, while updates are controlled by code executed speculatively.

We further observe we are not limited to finite state machines, but can support any model of computation where **updates** to any state are finite (i.e. can fit within the bandwidth constraints of the speculative primitive). This encompases Turing machines [118] as well as certain random access machines, which we investigate next. Figure 2.5 demonstrates the execution flow of a sample ExSpectre malware.

2.4.1 Turing Machine

To demonstrate that arbitrary computation can be performed cooperatively between the speculative world and real world, we implement a Turing machine, and configure it to run a 5-state Busy Beaver function [118, 17, 63]. This configuration allows us to run a large number of steps with very minimal logic.

Updates to this Turing machine are computed speculatively, while the real world keeps track

of the state and full tape of the machine. Thus, the logic of the machine is entirely contained in the speculative world, while the state may be externally visible (e.g. to a dynamic debugger). We note that the machine only operates when the trigger executes, making it difficult for an analyst with only access to the Turing machine to determine exactly what the machine will do from its initial state.

However, this toy example is meant only as an illustrative example of arbitrary computation, not as a robust means of obfuscation. Indeed, even the initial state of a Turing machine alone may reveal a significant amount of information. Furthermore, the analyst may attempt to locate potential speculative entry points, even without the help of the trigger program. We describe ways to address both of these next.

2.4.2 Unpacking and Decryption

While a Turing machine demonstrates that arbitrary speculative computation is possible, hiding malware this way has several drawbacks. First, Turing machines are a poor choice for practical computation, as they are inefficient and have no direct way to interface with the rest of the system (e.g. via system calls). Second, as mentioned, they leave a great deal of information available to the analyst, including the initial tape state, and a potentially small (enumerable) number of possible speculative entry points.

We explore a more practical application of using ExSpectre to perform **decryption** speculatively. To hide keys from the analyst, the key and decryption code only occur in the speculative world, while the initial payload program contains only the ciphertext. While partial plaintext will be available in the real world during execution, we emphasize that this only occurs when the trigger runs. Before this, the state of the program reveals only the ciphertext that will be decrypted. While the speculative entry point enumeration attack could be used to reveal the keys used to decrypt this ciphertext, we describe a way to derive the decryption key entirely from the trigger program. Thus, an analyst that only has access to the payload program will be unable to learn the key or decrypt the embedded ciphertext.

We also note that even when the trigger runs, decryption does not occur outside the speculative context, meaning that any traditional traps or debugging breakpoints placed on decryption instructions or routines will not occur, even as they are used speculatively. These instructions could even be obfuscated themselves by placing them in other misaligned instructions, and choosing a speculative entry point that jumps to the middle of other instructions.

We note that 200 instructions is too short for most software-implemented cryptography. However, modern Intel CPUs provide hardware support for AES, which we find only takes a handful of μ-ops to perform the instructions needed in AES decryption. We discuss details of our speculative AES decryption in Section 2.6.2.

2.4.2.1 Obfuscating keys with nested speculation

As mentioned, even with encryption, an analyst that can locate the speculative entry point and discover the decryption key. For instance, the analyst could locate the speculative entry point by searching for AES-NI instructions in the payload program, ultimately discovering the keys it derives and uses.

We can overcome this by having the trigger program communicate the decryption key to the payload program via the branch predictor. While prior work has used the branch predictor to exfiltrate keys from other sensitive processes [2], we inject a key into the speculating payload program from the external trigger program. To do this, we use multiple speculative entry points, each that derives a unique decryption key before calling a common decryption routine. Since the exact speculative entry point is determined by the trigger program, an analyst cannot trivially discover the decryption key directly from the payload program.

Still, an analyst could enumerate all potential entry points, testing each one until they find one that correctly decrypts the ciphertext. In a 1 MB binary, there are (at most) only 1 million possible entry points, providing just 20 bits of security, trivial for an analyst to brute force. An analyst simply needs to test each of the 2^{20} entry potential entry points to discover the correct key.

To increase security, we instead use nested speculation to **chain** entry points together. Rather

than derive the key from a single entry point, we have each potential entry point perform another indirect jump that the CPU cannot immediately resolve, forcing it to speculate while already executing speculatively. In other words, in the speculative world, we make an indirect jump that depends on a cache-evicted variable, prompting the CPU to double-speculate. The predicted target of that jump will also be driven by the trigger program's (mis)training of the indirect branch predictor. On CPUs that support double (or arbitrarily nested) speculation, we can repeat this process, with each new subsequent entry point determined by the trigger program. At each entry point, we shift in additional bits to a register as the AES key. Without the trigger program, an analyst cannot determine the path the payload program will take speculatively.

As an illustrating example, imagine a trigger program makes 30 training jumps, followed by 10 additional indirect jumps, and the payload program performs the same 30 training jumps before a stall. At this point, the CPU will predict the payload program will also perform the next 10 jumps, speculatively following the pattern of the trigger program.

If each nested speculative jump has the potential to land in 4096 (2^{12} possible locations, each entry point can shift in 12-bits to the key, for a total of 120-bits over the 10 jumps before calling the common decryption routine. A key constructed in this way would be infeasible for an analyst to brute force, as the payload program yields no information about which of the potential 2^{120} keys will be derived.

We describe our implementation of nested speculative execution in Section 2.6.2, where we speculatively derive a 128-bit AES key. Figure 2.6 demonstrates this in practice, the trigger program running a series of training jumps followed by indirect jumps influences the payload program to follow the same path.

2.4.3 Emulation

To combine our encryption and arbitrary computation in an efficient way, we implemented an emulator that gets its instructions from the speculative decryption described previously. In the payload program, the emulated instructions are initially encrypted under a key that will be

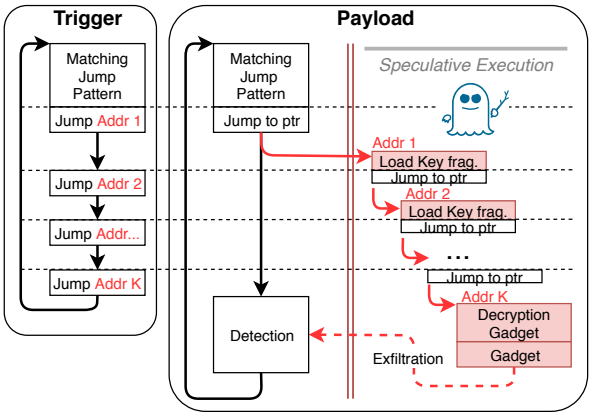

Figure 2.6: **Nested Speculation** — Some CPUs support nested speculation, allowing the branch predictor to speculate a branch while already executing speculatively. We use this to obfuscate **key derivation**. The trigger program executes a sequence of indirect jumps, which the payload program will follow speculatively. Each jump target in the payload program will add a small number of bits to a speculatively-computed key. Without knowing the exact pattern of jump targets (specified only in the trigger program), the analyst will be unable to determine the key when a sufficient speculative depth/number of targets is used. In our implementation, we used a speculative depth of 16 with 2^8 targets to derive a 128-bit key. While the *decryption gadget* may be easy for an analyst to find, without the key, the encrypted data remains inaccesible.

delivered by the trigger, as described previously. Once the trigger executes, it will cause instructions to be decrypted and run by the emulator.

Traditional reverse engineering methods will reveal only that emulation is being done, while the program being emulated remains encrypted. Even when the trigger is running, only the parts of the code that execute would be revealed to a careful analyst observing the CPU's committed state, while the remainder of the emulated program would remain hidden.

We design a custom emulator and instruction set—SPASM (Speculative Assembly)—that accommodates the constraints of our speculative primitive. SPASM is a 6-bit Instruction set, where all instructions (including operand, registers, and arguments) fit within 6-bits. This allows each step of the speculative world to emit a single SPASM instruction to the real world for emulation by a light-weight SPASM emulator. Using SPASM, developers can write programs, assemble and encrypt them into a payload program. When the associated trigger program runs, the payload will decrypt SPASM instructions in the speculative world, and execute them one at a time.

While the custom emulator that we developed gives higher level abstraction to an author, it still requires programs to be written in a custom assembly language. We note that the ExSpectre

model is not intrinsically linked to the SPASM emulator. A wrapper could be implemented around other existing emulators to construct instructions incrementally through the fixed-width channel (e.g. using 4 8-bit reads to reveal a single 32-bit ARM instruction), allowing for encrypted payloads to be written in higher-level languages. We also note that this provides flexibility to the authors, allowing them to completely redefine instructions or use a different instruction set altogether to hamper detection.

2.5 Triggers

So far, we have described using a custom program as a trigger, which performs a pattern of indirect jumps to mistrain the indirect branch predictor, leading the payload program to its speculative entry point. In this section, we describe alternative triggers, including using benign programs already on the system, and recent Spectre variants.

2.5.1 Benign Program Triggers

Custom trigger programs that are installed with malicious payload programs may be easy for an analyst to pair up and analyze. As an alternative to trying to hide the trigger program from the analyst, ExSpectre can use **benign** programs already installed on the system as a trigger.

For example, if a benign application makes a series of indirect jumps—thus training the indirect branch predictor—an ExSpectre payload can make similar indirect jumps leading up to its speculative entry point. The payload's speculative entry point will be determined by the benign application, but may be even more difficult for an analyst to discover, as now the ExSpectre trigger could be **any** application running concurrently on the system.

OpenSSL We experimented using the OpenSSL library as a potential benign trigger application, as its source code has a gratuitous use of function pointers which compile to indirect jumps. In addition, it has many complicated code paths that can be easily selected by remote clients through their choice of cipher suite. This allows a remote attacker to trigger ExSpectre malware on a server running a (benign) TLS stack supported by OpenSSL, simply by making a

large number of TLS connections with a specific cipher suite. We describe our implementation using OpenSSL as a trigger in Section 2.6.4.

In addition, an adversary could use a benign application (like OpenSSL) to **communicate** information covertly to the malicious payload program. For example, with OpenSSL, the attacker could have a pair of uncommon cipher suites, where using one results in communicating a 1-bit, while use of the other communicates a 0-bit to the payload. To receive data, the payload would have to do indirect jump patterns corresponding to OpenSSL code for processing both cipher suites, with the corresponding speculative entry points shifting in the appropriate bit. Thus, an adversary can communicate remotely (over a network) to the payload program indirectly via a benign application.

We observe that communication could also go in the other direction: from the payload program back to the remote adversary, also via the benign application intermediary. The payload program could influence the performance of the benign application, and the adversary could time responses from the benign application to receive covert information from the malicious payload [110]. This would allow the malicious payload to operate **entirely speculatively**, without assistance from the real world for keeping state.

2.5.2 Speculative Buffer Overflow

In addition to using separate trigger programs, ExSpectre can also use **trigger inputs** to a payload program to initiate its malicious behavior. While existing fuzzers and symbolic execution tools can discover traditional input triggers, we can leverage other Spectre variants to obfuscate our triggers.

To do this, we use Speculative Buffer Overflows (SBO) [76] to redirect control flow to a speculative entry point specified by user input. We design a payload program that takes arbitrary user input and performs appropriate bounds checks to ensure no traditional control flow violations could be exploited. However, using the Spectre 1.1 variant, control flow can still be violated speculatively, allowing the adversary to force a speculative entry point based entirely off the input provided. This allows the trigger to be an input, potentially even provided over a network if the

payload program accepts network data. Traditional symbolic execution and code coverage fuzzers will be unable to discover this trigger input, as they do not model the speculative state of the CPU.

To create an SBO-triggered payload program, we make the following code pattern, as seen from Kiriansky and Waldspurger's Spectre 1.1 description [76]:

```
if (y < lenc)
    c[y] = z;
```

With user controlled `y` and `z` and sporadically uncached `lenc`, an attacker can *speculatively* overflow array `c` to overwrite a return address (or function pointer, etc) and redirect control flow. Note that the bounds check on `y` will ensure that this program will not actually allow a buffer overflow to occur, but the attacker can nonetheless use this to influence a speculative entry point based on their choice of `y` and `z`.

We make use of this pattern in a willing payload such that user input can intentionally mistrain the branch predictor by repeatedly sending valid (in-bounds) values of `y` before sending a value that would overflow the bounds of `c`. The speculative entry point is also chosen by the trigger in this scenario as `z` contains the address of the speculative entry point, allowing the attacker to create a ROP-style speculative execution path through the payload.

We implemented an experiment to determine the number of times a branch needs to be "trained" before it can be used as a speculative buffer overflow. Figure 2.7 shows that several hundred benign inputs are needed to reliably be able to observe speculative buffer overflow behavior.

2.5.3 Speculative Store Bypass

Speculative Store Bypass (SSB) (Spectre variant 4) can similarly be used to construct an internal (input-based) trigger using the CPU's speculative load-store forwarding [88]. In a speculative store bypass, the CPU incorrectly speculates that a store will not alias with a future load, and uses a stale (wrong) value for the result of the speculative load.

To redirect control flow, a payload program could use a function pointer or indirect branch

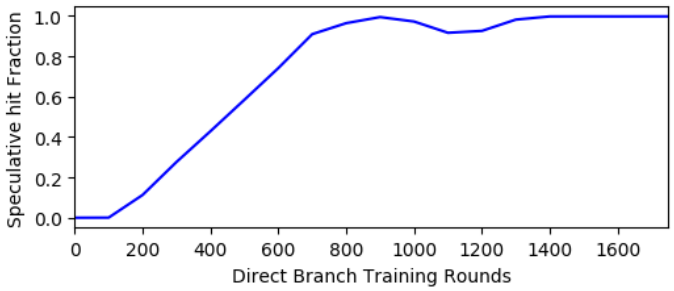

Figure 2.7: **Speculative buffer overflow warm-up** — The direct branch predictor must be trained to expect that a branch will go a specific way before speculative buffer overflows can be used. We varied the number of times a branch was trained to be taken and observed the fraction of times we achieved a speculative buffer overflow execution immediately after (measured by observing if a speculatively-loaded value was present in cache avgeraged over 20,000 trials). We find that a branch must be trained in a direction hundreds of times before it can be reliably used in a speculative buffer overflow.

target register as the destination for a speculative store bypass, causing the CPU to use a stale value to speculatively determine where it would go. The stale value could be controlled by a previous unrelated input, allowing an adversary to specify the speculative entry point in a carefully crafted data input. While the program never executes at this stale address in reality, the CPU will briefly speculatively execute there, enabling ExSpectre payloads. Like the speculative buffer overflow, this trigger also allows ROP style chains to execute a series of speculative gadgets.

2.6 Implementation and Evaluation

In this section, we discuss implementation details for our payload and trigger programs.

2.6.1 Turing Machine

We designed our Turing machine implementation to work with our custom trigger program, with 28 indirect jumps mimicked by the Turing payload program. We implemented a 2-symbol 5-state Busy Beaver Turing machine logic at the speculative entry point (in 42 x86-64 instructions), returning the state update, symbol to write, and tape move direction in a single byte via a cache

side channel.

We observed in our implementation that it is important that all values used in the speculative world—as well as the code itself—be cached. If these are evicted, the speculative code may fail to run, or the CPU may **speculate** on the value of the uncached item, which may be incorrect. While this does not impact the correctness of normal programs whose incorrect speculations will be resolved, our speculative code reports results back to the real world before this resolution. In our Turing example, we observed this as incorrect state transitions.

This error is particularly devious, as it is not an error of bit flips or noise, but rather the processor speculating what the speculative gadget will read from memory. Thus, error correcting codes on the reported result do not improve the situation.

Instead, we repeat the execution several times and look for the modal value over all iterations. We measured the error rate of our implementation as a function of how many redundant iterations of the same step, and found that 10 redundant iterations resulted in 1 error every million Turing steps, with the error rate dropping exponentially as iterations increase. We choose 11 iterations as a conservative bound (error rate measured to be 0), and computed 1 million Turing steps at a rate of 1351 steps per second.

2.6.2 AES Decryption

The speculative world is able to take advantage of the AES-NI instructions to decrypt messages. However, the speculative upper-limit of about 175 μ-ops is not enough to allow us to compute the key expansion, even using the `aeskeygenassist` instructions. To avoid this, we can either preload the expanded key schedule into the program (instead of the key), or use a cheaper (non-standard) key expansion algorithm. For the former, we note that an analyst could observe the structure of a normal key schedule, but we can avoid this by simply selecting 11 random round keys. We note that this should not weaken the security of AES, as we can ensure the round keys are not linearly related.

We wrote our AES decryption payload in 35 x86_64 instructions and 2 lines of C (which

compiles to an additional 31 x86_64 instructions). The payload implements AES-CTR mode decryption, reading a global index and returning the decrypted byte at that location in the ciphertext via the cache side channel. In this model, the speculative function decrypts a full 16-byte AES block each iteration, but only returns the bits specified by the index.

We demonstrate the speed that information can be decrypted via the speculative world, and we vary the channel width of the side channel from 1 to 12 bits to measure its performance. At low channel width, reading from the cache side channel requires timing reads from only 2 locations, while at 12-bits, the side channel requires reading 2^{12} locations. On the other hand, there is a fixed overhead per speculative iteration that favors increased channel width to maximize bandwidth. As shown in Figure 2.8, 8 bits is the optimal side channel width, allowing us to decrypt over 5,000 bits per second (625 Bytes/sec). We note an improvement over this rate by loading multiple values into the probe array during speculation. Instead of a single probe array of 1024 entries communicating 10-bits, we can split the array into four sections of 256 entries each, and signal four times (one per section) during speculation. This provides a 32-bit channel overall, while still only having to probe 1024 entries. This method is capped by the limited μ-operation budget, however our implementation using these parameters (four sections of 256 entries) is able to decrypt over 11 Kbps (1,425 Bytes/sec).

We also implemented our nested speculation technique for obfuscating keys, making 256 speculative landing spots that each shift 8 unique bits into the 128-bit register `%xmm0`, and then performing an indirect jump. We then had a custom trigger program perform 16 indirect jumps (after the initial 28) that corresponded with 16 randomly-chosen landing spots in the payload program, training the branch predictor. When the payload program reaches the first speculative jump, it follows the same pattern speculatively, eventually filling `%xmm0` with the corresponding $16 * 8$ bits. We then used the `aesenc` instruction to expand these 128-bits to a full key schedule, and performed decryption as described previously. Thus, without the trigger program, an analyst has no information about what key is used to decrypt the ciphertext in the payload.

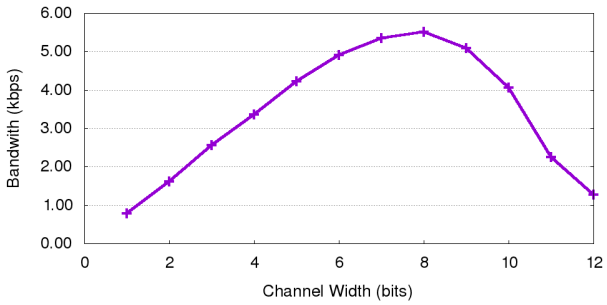

Figure 2.8: **Speculative Bandwidth** — Using our speculative primitive, 1KB of data can be decrypted and exfiltrated at a speed of 5.38 Kbps from the speculative world with 20 redundant iterations per round (to ensure correctness). Increased channel width exfiltrates more data per round, but takes longer to measure the cache side channel. Optimal throughput is achieved with an 8-bit channel.

2.6.3 Emulator

We have implemented our custom instruction set architecture—SPASM—as a model using two pseudo-registers, and 6-bit instruction length which allows for a relatively direct programming model in which structured values can be entered into memory locations before making a systemcall.

In this model of computation there are effectively no instruction arguments, as we must return an entire instruction from the speculative world inside the limited-width cache side channel. Although other small instruction sets exist, they either allow variable instruction lengths, are too long even in reduced form, or did not have significant support to make them favorable for developers.

We used 6 bits in the construction of this instruction set as our goal is to limit the length of each opcode as much as possible. Note that this is different from the goal in maximizing bandwidth, as our goal now is to maximize instruction throughput. Given our short instructions, loading values into registers requires shifting in 4-bits at a time. SPASM has two registers that act as a pointer and working register, that can be used to perform jumps, arbitrary memory reads and writes, and basic arithmetic. We also have a `syscall` instruction that makes a real system call to the underlying operating system with parameters loaded from the SPASM state, allowing us to interact with the real world.

In SPASM we have implemented multiple example programs that we encrypted and loaded

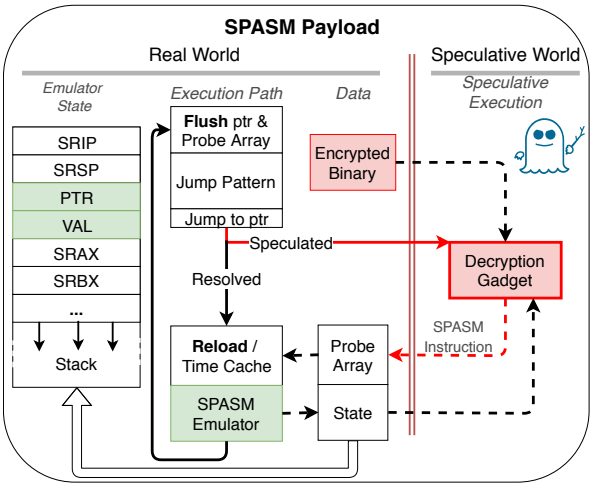

Figure 2.9: **SPASM model** — Our SPASM emulator speculatively decrypts instructions, and emulates them in the real world. The *Speculative Computation* decrypts the encrypted SPASM binary using AES, returning the result through the side channel to allow the *Real World* to update the emulated state and make system calls on behalf of the speculative world.

into a ExSpectre payload, which decrypts and emulates SPASM instructions only when the corresponding trigger program is running. We have implemented a *HelloWorld* program that prints to `stdout`, and a *FizzBuzz* program that demonstrates control flow and arithmetic operations while printing to stdout. Finally, we implemented a *ReverseShell* program that opens and connects a TCP socket to an attacker-chosen location before executing a local shell and allowing the remote adversary to issue shell commands on the victim machine. Figure 2.9 details the high-level flow of a SPASM payload.

Our *ReverseShell* program consists of 355 SPASM instructions, and makes six system calls to open a socket, connect to it, duplicate I/O file descriptors, and perform an `execve` system call to open a shell. In our tests using 5 iterations per decrypted instruction, the *ReverseShell* program takes just over 2ms to launch a reverse shell once triggered.

2.6.4 OpenSSL Trigger

To demonstrate a benign trigger application, we implemented an ExSpectre payload that would trigger when running concurrently with OpenSSL. We disable ASLR for simplification, but note that branch predictors can also be used to determine ASLR offsets of co-resident applications, and our attack adjusted accordingly [38].

We used `gdb` to run an instance of an OpenSSL server (version 1.0.1f), and printed out every

instruction executed and its address after a breakpoint on the `SSL_new` function. We then made a TLS connection to the server, which produced over 13 million instructions, including over 359,000 direct jumps and 28,000 indirect jumps. We then searched for the longest repeated set of more than 28 indirect jumps that ends with a unique jump (i.e. source and destination do not occur in the previous 28+ indirect jumps).

We discovered a candidate that corresponds to code in OpenSSL's `nistp256.c` that contained 31 indirect jumps repeated 254 times each handshake. This code is used during the TLS key exchange as the server computes the ECDHE shared secret. We made a list of 31 source-destination address pairs for these indirect jumps, and constructed a `jump/ret` chain to mimic the same jump pattern in our payload program. Our payload program mimics the first 30 indirect jump source/destination pairs, with a final jump going to a cache timing function in our payload program. However, due to the prior pattern, this last jump is frequently mis-speculated (about 3.5% of the time), and instead goes to the destination corresponding to the 31^{st} jump in OpenSSL, which serves as our speculative entry point.

We ran experiments on an Intel Haswell i5-4590 CPU, with OpenSSL and our payload program pinned to the same core using `taskset`. We induced the jump pattern in OpenSSL by running Apache benchmark against it to generate thousands of TLS connections using the ECDHE key exchange with the secp256r1 curve (ECDHE-RSA-AES256-GCM-SHA384). When running Apache benchmark locally, our payload program reliably executes (speculatively) at the intended speculative entry point about 3.5% of the time. When apache benchmark runs on a remote machine, this rate drops to approximately 2.0%. Nonetheless, these are both sufficient to perform computation, as our payload can simply increase the amount of iterations needed to extract meaningful results from the speculative world.

We verified that our payload program did not execute at the speculative entry point when we ran other programs that simply consumed CPU on the same core. In addition, when we used Apache benchmark to create thousands of connections with a different cipher suite (DHE-RSA-AES128-GCM-SHA256), we similarly saw no speculation at the entry point. This could allow an

adversary to use an obscure or uncommon cipher suite to trigger a malicious ExSpectre payload program on a remote server.

2.7 Discussion

2.7.1 Defenses

We now address possible defenses to detecting and reverse engineering malware that uses ExSpectre.

2.7.1.1 Implemented Mitigations

Multiple patches and micro-code updates have been developed to mitigate Spectre vulnerabilities, however, none of these entirely prevent ExSpectre malware from working, as they are generally not designed to protect programs that willingly use Spectre against themselves.

Indirect Branch Predictor Barrier IBPB is used when transitioning to a new address space, allowing a program to ensure that earlier code's behavior does not effect its branch prediction. IBPB requires CPU and operating system support. However, we observe on Linux that processes running under the same user group do not receive IBPB protection, enabling ExSpectre when the trigger and payload run under the same group. Furthermore, IBPB does not prevent the speculative buffer overflow variant of ExSpectre described in Section 2.5.2.

Single Thread Indirect Branch Predictors STIBP prevents sibling hyperthreads from interacting via the indirect branch predictor. However, this does not prevent co-resident processes from cooperating when they run on the same logical core.

Indirect Branch Restricted Speculation IBRS prevents code in less privileged prediction modes from influencing indirect branch prediction in higher privileges (e.g. the kernel). This does not prevent speculative execution in a willing payload program in a less privileged speculation mode.

Retpoline is a software mitigation that replaces indirect jumps with a special call/overflow/return sequence, controlling where the CPU will speculate the indirect branch to a contained (and benign) section [119]. However, this defense is opt-in which ExSpectre binaries could simply choose to not use, or alternatively use the unaffected speculative buffer overflow variant.

2.7.1.2 Malware Detection

While not a primary goal of ExSpectre, we consider the ability of ExSpectre malware to hide from detection.

When using the cache side channel variant of ExSpectre, the payload program must at least occasionally watch this side channel, offering a potential method for detecting ExSpectre malware. Analysts could search for telltale signs of cache inference behavior, such as the use of `clflush` instructions or reading cycle timings. At the cost of performance, ExSpectre could choose to use a more subtle cache side channel that does not require this, such as `Prime+Probe`, or by exploiting race conditions between multiple threads to allow the speculative world to influence the behavior of the real world.

ExSpectre could also use another side channel method that avoids the cache to exfiltrate information from the speculative gadgets, such as the branch predictor itself [40], memory bandwidth, power utilization, or contention over other shared resources [76]. While cache channels tend to have the highest throughput, they are not the only resource that must be monitored to detect or prevent these types of attacks.

Anti-Virus Detectors We verified that modern Anti-Virus technologies were unable to detect and flag ExSpectre malware. We used ClamAV, BitDefender and rkhunter, which mainly rely on signature and string based detection. BitDefender does feature support for unpacking or extracting malware, though appears to simply try unpacking using several known packers and encoding formats [1]. Thus, it is not surprising that these tools cannot detect ExSpectre.

Bare Metal Modern malware often uses hardware minutia to identify and fingerprint execution environments in order to detect when it is under debugging or inspection [82, 8, 98].

To prevent such identification, analyzers often employ "bare metal" execution [74], running the malware on dedicated hardware that allows introspection and observation of the system without interfering with its normal operation. This prevents malware from using so-called "red pill" checks to observe that it is under test (and hide its malicious behavior) [75]. However, to the best of our knowledge, no publicly available bare-metal environments allow introspection on the speculative state of the CPU, making it difficult to analyze ExSpectre malware. However, such environments could be useful for observing the behavior of ExSpectre malware in the presence of its trigger if available, as modifications in the real world could be easily tracked.

Symbolic execution has also been used to find environmental red pill checks [109]. However, such analysis would be ineffective against ExSpectre, as symbolic execution does not reason about speculative paths and how they might influence a program.

2.7.1.3 Reverse-engineering triggers

For program-based triggers, an analyst could attempt to find the trigger program by examining the execution path of the payload program, and locating a common indirect jump pattern between payload and potential triggers. Since both programs must share a common indirect jump pattern to interact via the indirect branch predictor, there must be some overlap which is unlikely to occur randomly between two programs.

We note that while the analyst may learn the execution path (and thus true indirect jump pattern) of the payload program, they may not be able to capture every potential execution path in all potential triggers. For example, in the OpenSSL trigger, the analyst may not have captured all potential indirect jump patterns, as doing so would require exhaustively connecting to OpenSSL with different cipher suites, extensions, and failed handshakes. However, the analyst can still make a list of indirect jump locations in a suspected trigger program, comparing these to the jumps taken by the payload. If there is significant overlap, the analyst could spend time to discover what inputs to the trigger program produce similar indirect jump patterns, thus discovering the trigger.

ExSpectre malware could attempt to thwart this analysis by using decoy indirect jumps that

do not correspond with the trigger, but potentially correspond with other (non-trigger) binaries. In addition, this analysis method is ineffective at inspecting the speculative buffer overflow variant described in Section 2.5.2, as it does not use a separate trigger program.

Alternatively an analyst may attempt to identify sections of the program or dead code that will be used to access the probe array and thereby find the speculative gadgets. However, identifying sections of memory that will access the probe array is equivalent to the "Must Alias" or "Points-To Problem" which has been proven undecidable without significant restrictions [103, 79].

2.7.2 Future Work

ExSpectre demonstrates a general model for hiding execution in the speculative world and examines the implications and limitations on modern processors. Given the wide-spread nature of the Spectre vulnerability and the ubiquity of side-channels, we believe that this work can be directly extended to other architectures, such as ARM, and other processors making use of speculative branch prediction.

2.7.2.1 Multiple Triggers

To create further difficulties for an analyst, or to further target the execution environment, it is possible to have the payload program to combine multiple triggers. Instead of requiring only a single trigger program, the payload could require multiple trigger programs to be running simultaneously, or in a particular order. Alternatively, the payload could combine trigger programs with input triggers, forcing an analyst to understand multiple variants simultaneously.

This could allow fine-grained targeting of malware. For instance, the attacker could distribute trigger programs through different channels to target different sets of victims, and have the ultimate payload only operate at the intersection of these groups. As an example, one trigger program could be distributed to a particular country (e.g. Iran), and another to a particular device globally (centrifuge controllers), resulting in the malicious payload (Stuxnet) only being revealed and executed on the intersection of these two groups.

2.7.2.2 Virtual Machines

Virtual environments could also be host to ExSpectre malware and triggers. For instance, malware on one EC2 instance could potentially be triggered by a trigger program on another seemingly unrelated instance. We have found that the hypercall context switch from guest to host on VirtualBox is lightweight enough that a trigger program running in a guest can activate a payload program running in the host on the same CPU core. However, we have so far been unable to go in the opposite direction, and similarly have yet to achieve guest-to-guest interference. More work is needed to determine if such barriers are possible to overcome, and if stronger isolation is needed in the virtual machine context.

2.8 Related Work

2.8.1 Weird Machines

ExSpectre shares many properties with *weird machines*—a machine which takes advantage of bugs or unexpected idiosyncracies in existing systems to perform arbitrary computation [14, 15]. In particular ExSpectre showcases the ability to use CPU speculation to compute.

Recently there has been a trend of features in modern processors such as multiple threads sharing system resources and optimizations done across the process isolation boundary which lead to opportunities for "weird machines" [21]. In particular, previous examples of "weird machines" include traditional vulnerabilities such as buffer overflows, format string exploits and return oriented programming [96, 55, 111]. Weird machines have also been built using operating system page faults, enabling the computation of arithmetic and logic operations without the use of traditional instructions [9]. ExSpectre extends research in "weird machines", and takes advantage of speculative execution to execute instructions that otherwise appear to be dead code.

2.8.2 Covert Channels

Spectre builds upon prior work on cache side channels, and similarly uses them to leak information from processes [99, 143, 97]. In ExSpectre, we use the branch predictor as a **covert channel** [78] between the trigger program and malware payload, allowing the malware's (speculative) execution path to be influenced by the trigger.

Previous work has examined how to share information over covert channels, such as across virtualized environments on cloud systems [135], using L1 and L2 cache to share information [99], measuring temperature to create a thermal covert channel [86, 10], and taking advantage of processor architecture to leak information [126]. This includes using the branch predictor itself as a covert channel [39, 37], which ExSpectre similarly uses.

However, we note that the covert channel used in ExSpectre need not involve two cooperative programs, and we demonstrate using the benign OpenSSL as a non-colluding program involved in utilizing this covert channel.

2.8.3 Speculative Execution

ExSpectre builds on Spectre [77] and Meltdown [84] which leverage speculative execution to leak sensitive information from vulnerable processes. Follow up work has identified several new Spectre variants, including speculative buffer overflows and speculative store bypass [76, 88], and has investigated additional ways to leak information using branch predictors as a side channel [40]. Researchers have also leveraged Spectre and speculative execution more generally to demonstrate web-based vulnerabilities [54, 110] as well as to leak control flow, keys, and other information from the hardware isolation provided by Intel SGX [95, 18, 16, 80]. Spectre has additionally been proposed as a way to thwart taint tracking by using speculative execution to copy data between buffers [61]. ExSpectre likewise takes advantage of speculative execution, but with the goal of hiding arbitrary computation from reverse engineering, rather than extracting secrets from vulnerable programs. ExSpectre also benefits from new Spectre variants: as we showed, speculative

buffer overflows ("Spectre 1.1") can be used as an alternative trigger for malware.

2.9 Conclusion

We have presented ExSpectre, a model for hiding computation in speculative execution that is fundamentally different than existing methods of code obfuscation. Through a series of experiments we have classified the capabilities and limitations of this speculative primitive and demonstrated various example applications. We have demonstrated the potential of using speculative execution in several applications, including a Turing machine, SPASM emulator, remotely-triggered payloads, and AES decryption. We have also examined Intel's responses to Spectre and Meltdown and noted how their defenses affect ExSpectre and how further variants of Spectre can be adapted to ExSpectre. Current analysis techniques for reverse engineering are insufficient to reason about the behavior of these programs.

Ultimately, silicon and microarchitecture patches will be needed to secure CPUs against this kind of malware. Until then, attackers may iterate and find new variants of ExSpectre-like malware. In the meantime, new detection techniques and software-level mitigations are desperately needed.

Chapter 3

Refraction Networking

At the upper end of large scale systems is the internet and the vast array of networked devices of which it is composed. At it's core, the internet facilitates communication and allows systems to interoperate remotely. Given the scale and complexity of the internet it is not surprising that unintended functionalities arise in the TCP/IP layers responsible for routing traffic and moderating sessions. In this chapter specifically we examine refraction networking, which allows proxy operators to partner with ISPs to move proxy logic to the middle of the network. This increases the stakes of IP blocking for censors as an entire arm of the internet can be used to complete a connection and the censor must choose between allowing access and cutting off that arm.

Originally proposed around a decade ago refraction networking has seen multiple proof-of-concept implementations and more than one deployment. The initial refraction protocol to reach a test deployment was TapDance, whose novel design specifically operated on mirrored network tap traffic, forgoing support for inline blocking in favor of deployability. However, the TapDance deployment illustrated that there were significant drawbacks in the TapDance protocol [49, 121]. This motivated the development of the Conjure protocol [50] reducing the burden on background sites while at the same time increasing the flexibility to resist censorship. Like TapDance, Conjure has now seen a significant deployment spanning multiple ISPs and continuing to grow. However, in operationalizing and scaling this use of unintended functionality of TCP/IP we have encountered new challenges both technical and organizational.

In this chapter we reflect on our experience of operationalizing and scaling non-standard

functionalities for beneficial purpose as part of the production deployment of the Conjure refraction networking system. In general we find that in its current form refraction networking provides a buffer behind more direct censorship circumvention techniques. When censors take action against proxy traffic we see that IP and protocol based blocking that affects traditional proxies is ineffective against refraction networking. This buffer allows clients to stay online and continue to access the internet while the proxy operator works to shuffle addresses or wait out protocol over-blocking.

3.1 Introduction

In the last decade since the publication of the original proposals for refraction networking both refraction and the wider censorship circumvention communities have identified and risen to new challenges. This provides an appropriate moment for both reflection on the design choices and lessons of previous work and anticipation of future research efforts.

Refraction can learn from and incorporate techniques that the censorship circumvention community have developed over the last decade, while at the same time scaling for a potential large-scale deployment and leveraging the unique routing perspective to create a more open internet.

Over the last decade refraction has progressed from proposal, through research prototype, to deployed proxy with over 1.5 million daily users. We analyze the latest development in censorship strategies [115, 11] and proxy design including relatively novel techniques like domain fronting [44] and CDNBrowsing [64]. We then provide a brief survey of refraction networking schemes and lessons learned by looking at the first [137, 72, 68], second [136, 34, 13], and current [50, 112] generations of refraction networking. With this context we first propose potential engineering designs leveraging recent developments in censorship circumvention [59, 42] to grow and strengthen refraction deployments before examining broader future research direction.

The rest of this chapter is structured as follows: Section 3.2 will discuss recent censorship measurement efforts and introduce censorship circumvention tools, Section 3.3 will discuss historical refraction networking proposals and refraction specific routing attacks, Section 3.4 will explore opportunities for integrating novel censorship circumvention strategies into refraction deployments,

and finally Section 3.7 will provide a discussion on future research directions and potential investigative "jumping off points" for work in these areas before we conclude in Section 3.8.

3.2 Censorship Background

At their core proxy servers are hosts outside of a censors sphere of influence that make requests on behalf of clients, while using encrypted channels to share the results. This prevents censors from introspecting on the content of network traffic within their jurisdiction, making proxies a primary target for blocking. Proxies are discovered and blocked in a number of ways. A primary strategy is to interrupt the DNS system and prevent clients from learning proxy addresses. Investigations into this type of censorship have shown that censors rely on national network infrastructure to enable censorship strategies [138, 104]. While DNS plays a large role in many censorship strategies, this work focuses more closely on proxy transports. Here we provide a brief and non-exhaustive survey of the latest and most effective censorship strategies, circumvention proxy designs, and probe resistant transports.

3.2.1 Proxy Discovery

A 2020 study by the Censored Planet team identified DNS, HTTP(S), TCP/IP interference among the most widely deployed censorship strategies globally [115]. This longitudinal study observed that censorship typically comes in bursts surrounding specific events like elections or political unrest. Interference in HTTP is typically carried out by passively inspecting content in search of restricted keywords, detection of which causes censors to interfere by tearing down connections or redirecting to block pages. HTTPS blocking relies heavily on the server name indication (SNI) TLS extension and allows censors to block domains that are known to host censored content. In the absence of effective DNS or HTTP(S) blocking censors block by IP address, typically preventing server-to-client traffic. During the 20 month study the authors identified censorship events in 15 unique countries on top of the long-term censorship strategies deployed in countries like Iran, Turkmenistan, and China.

In order to block proxies that don't provide explicit signals like domain names censors must first identify them. To do so censors are able to impersonate users, monitor large volumes of traffic, and interfere in suspected proxy connections. However, network interference can be expensive as business outages and political backlash can cost governments significant time, money, and influence. In order to establish high confidence that a particular server is in fact a proxy endpoint censors employ a number of techniques. Shadowsocks is an open source proxy that has seen wide adoption throughout China. A 2020 investigation into the measures that the Chinese government takes to block Shadowsocks servers found that a combination of deep packet inspection (DPI) and active probing allow the Chinese great firewall (GFW) to establish confidence before blocking hosts [11]. The GFW would watch for specific packet formats to cut down the initial set of suspicious hosts. They would then send a set of probes in an attempt to illicit specific responses from a suspected server, including replaying portions of observed sessions. This process ostensibly provided both confidence that a server was a proxy as well as an indication of the implementation in use (as there are multiple versions of Shadowsocks available). From here proxies would be added to a block list that dropped all server-to-client traffic matching either the proxies IP address, or IP address and port. To clients this would look as though the server had gone offline. This strategy demonstrates both passive and active censorship strategies as the preliminary step relies on traffic monitoring to *passively* filter as much classifiable benign traffic as possible before *actively* engaging or interfering with the proxy host and its client connections. While active probing is typically a more aggressive strategy, china has been seen sending probes as far back as 2011 [35] and continues to do so to this day.

While not yet widely deployed, strategies have been proposed to identify traffic based on behavioral indicators like packet size and inter-packet timing [30, 31]. These strategies seek to identify hosts based on the side-channels in network protocols that are difficult to obscure. Fortunately these traffic analysis strategies typically require a lot of state to be kept and result in high false positive rates (given the relatively low base rate of proxy traffic) [71]. Due in theory to this relative risk of blocking benign hosts and the high cost of deployment censors have yet to be caught

using such a system.

3.2.2 Proxy Design

Many modern proxies are designed to circumvent these censorship efforts. To address active probing by censors many proxies require a secret distributed to clients to be present in connections. While this does not prevent a censor from impersonating a client and connecting to the proxy it does prevent proxy enumeration through broad scanning efforts that illicit distinguishable responses from proxy hosts. This can include probes as generic as TCP buffer boundaries and timeouts which were found to fingerprint many proxies with relatively high accuracy [49].

The probe resistant proxies must then resist the traffic analysis that would distinguish their protocols from benign traffic on the internet. Strategies for doing so can be broadly classified into one of three categories. First are protocols that *mimic* other protocols by hiding in underlying data channels or hollow out legitimate traffic to replace it with proxy data [89, 130]. A common pitfall for mimicry protocols is behavioral analysis; for example, a video-call platform does not provide the same traffic patterns as the general internet usage that a mimicking proxy might display [66].

The second class of proxies *randomize* their traffic to look unlike any other protocol on the internet. While a large majority of traffic on the internet uses well defined and well known protocols, there is a long tail of obscure protocols, some of which control important industrial infrastructure. Randomizing proxies attempt to take advantage of this. However, where mimicking protocols struggles by not looking similar enough to the benign traffic that they imitate, randomizing protocols struggle to prevent any classifier from uniquely identifying their traffic. For example aggregated information about packet entropy can work against randomizing protocols over multiple packets or multiple connections.

The final class of proxies *tunnel* traffic through hosts that handle large volumes of benign traffic. This can be done by relying on generic domain name assigned by cloud providers and popular protocols like TLS to blend in with high value traffic. Meek does this by using TLS connections to content distribution networks (CDNs) and cloud providers that allow SNI to contain

a generic domain name (e.g. `google.com`) while the encrypted HTTP header directs traffic to a well specified host [44]. While this leverages the utility of cloud providers as a bargaining chip to prevent blocking, it comes with significant overhead. Network services in cloud platforms can be expensive, and domain fronting proxies are always at the whim of the cloud platforms who generally try to avoid missing out on large markets in censored countries due to blocking . For this exact reason Domain fronting has lost support from some the biggest cloud hosting platforms including Google's GCP, Amazon AWS, and most recently Microsoft Azure [41, 19, 25]. Cache Browsing proxies rely on a similar insight. Globally distributed CDN networks often have large rotating address spaces that censors are unwilling to block entirely — as much as 30% of the statics sites on the internet are served from CDN caches. This often allows static content that would normally be censored to become available by simply making that content available on a CDN [92, 64]. The connection to the CDN is encrypted and the name resolution does not reveal the content that users intend to access, requiring censors to block entire CDN networks when using traditional domain or address based blocking methods. Unfortunately this does not work for all content as CDNs primarily host static sites and moderate hosted content to a degree.

3.2.3 Tor & Pluggable transports

The Tor network operates as a system of proxies that provide guarantees about traffic anonymity. While anonymity is not the focus of this work the ingress nodes of the Tor network have seen some of the most aggressive censorship efforts and in turn the most advanced anti-censorship protocol deployments. The design of the Tor network requires that clients construct a path using a set of three proxy nodes. Clients select this set from a "consensus" list of currently available nodes published regularly by the Tor maintainers. This leads to a trivial enumeration attack in which all ingress nodes known as guards can be block-listed in censoring countries. To combat this Tor maintains a list of private nodes called *bridges* that are handed out in controlled manner.

As far back as 2011 it was publicly known that the GFW was probing internet addresses to identify and block bridges [35, 83], prompting the Tor foundation to identify probing as one of

10 key challenges facing the Tor network at the time [5]. To protect the bridges Tor developed a number of probe resistant proxy protocols called *pluggable transports* to wrap and secure first hop connections into the Tor network. Historically obfs4, a randomizing protocol designed to look like nothing has been reliable and effective at preventing bridge discovery. By design obfs4 requires clients to demonstrate knowledge of a secret shared out of band, any connections unable to demonstrate that knowledge receive no response from the server [141]. This aligns the challenge of distributing secrets with the challenge of distributing bridge node addresses. Tor relies on email, word of mouth, and Meek domain fronting to aid in distributing bridge details. While Meek is effective against many censors, it is prohibitively expensive to route bulk client traffic so the Tor nodes limits domain fronted use to bridge distribution efforts. A second pluggable transport seeing growing use is the snowflake system which relies on volunteers around the world to run WebRTC clients supporting peer-to-peer (P2P) connections out of censored countries [12]. To connect to a snowflake peer a client in a censoring country must first perform a STUN/ICE request to identify their public address before sharing a session description protocol (SDP) offer with the snowflake broker that matches peers together. This broker connection is once again fronted with Meek. The P2P nature of snowflake makes it resistant to probing as "snowflakes" are meant to be short-lived and transient and often require particularized connection tuples to satisfy holes punched in NAT networks.

Despite these efforts the Tor network remains blocked by the GFW. One of the largest contributing issues is the scale of the Tor networks usage measured against the relatively limited number of bridge nodes available. By abusing the channels for distributing bridge nodes to legitimate users censors are able to effectively enumerate all usable bridge addresses. While obfs4, snowflake, and Meek effectively circumvent the GFW, bridge enumeration prevents Tor from being functional for Chinese users [83, 27].

3.3 Refraction

Over the past decade we have seen many instances of censorship efforts, and one of the most common patterns throughout those events is the relative position of power that censors hold in national network infrastructure. Refraction networking in general proposes a response to that power where the censorship circumvention tools leverage similar positions in routing infrastructure. By moving proxy logic to the middle of the network we take on both unique challenges, and enable new censorship resistance.

Multiple refraction networking schemes have been proposed over the last decade with the connecting tissue being the threat model and high level construction. All refraction schemes deploy stations at or near Internet Service Provider (ISP) traffic ingress points. They require that the partner ISP is not malicious, and that a non-zero volume of traffic will transit the link that the station monitors. Some refraction schemes require the participation of downstream hosts [137, 34, 136], and assume that those hosts are non-malicious, but do not require that they voluntarily participate.

Each iteration of refraction networking has identified challenges, or proposed unique contributions that strengthen the next generation of proxies. Here we cover significant refraction works in what we consider to be the first three generations, illuminating both their strengths and the lessons learned.

3.3.1 First Generation

The first generation of refraction networking protocol design was a result of three independent parallel efforts examining a similar proxy design paradigm. These set the baseline for refraction protocols by proposing deployment at middlebox hardware, and leveraging that position against censorship efforts. All three relied on downstream hosts which they called decoys in some capacity. Karlin et al. proposed the *Decoy Routing* protocol which relied on sentinel messages embedded in TCP streams to indicate to an on-path station to hijack the TCP session and transition to a

proxy protocol [72]. This handshake required the station to close the session with the decoy host using a TCP `RST` and suppress any further traffic from that host to the client. Telex proposed a similar one-shot system in which clients would tag a TLS flow with an encrypted message in the ClientHello nonce field. The message would be indistinguishable from uniform random to anyone without the stations keys. This allowed the station to optimistically decrypt ClientHello nonces of TLS sessions that it observed and discover the master secrets of those TLS connections securely leaked by the client [137]. From here the station would be required to suppress traffic to and responses from the decoy host allowing the station to use the negotiated TLS channel for proxy traffic. Cirripede proposed a slightly different strategy of sending probes past a refraction station using covert channels inside of the TCP headers of legitimate traffic. Once this shared information was established the client would open a TLS connection to a benign decoy host which the station would hijack, intercepting traffic sent by client to prevent interference by the decoy host [68].

These early refraction proxy designs lay out many of the foundational concepts for constructing secure communication channels with middleboxes on the internet by parasitizing communications with downstream hosts. This set of proposals developed multiple tagging and signaling protocols and provided an early iteration of a multi-station refraction deployment. At the time these systems were all known as "decoy routing" protocols, which seems fitting given the reliance on downstream decoys. However, going forward decoy routing could be considered a sub-set under the umbrella term refraction networking as reliance on decoy hosts is not a pre-requisite.

3.3.2 Routing Attacks

In response to the early refraction proposals Schuchard et al. introduced an attack unique to refraction networking. Because the proxying logic is deployed at middle boxes in the network infrastructure and the internet is designed to be resilient to broken links, censors may be able to simply "route around" refraction stations [108]. To test the theoretical effectiveness of the routing around decoys (RAD) attack the authors simulate global network links based on CAIDA inferred network topologies and place refraction station stand-ins at random autonomous system

(AS) nodes. They claim that a routing capable censor can control traffic routes to detect 90% of refraction stations when deployed in as many as 4000 unique ASes. They then claim that as much as 10% of the internet could deploy refraction and censoring countries would only suffer a 2 - 10% reduction in global reachability based on alternative routes available to censors. Censors can further leverage this power to test suspicious connections by replaying TCP packets on an alternative path that routes around the suspected refraction station. A true connection will be uninterrupted, while a refraction connection will be terminated by a RST from the decoy. Finally the authors observe that a censor will, under certain conditions, be able to force an asymmetric route between the client and decoy. For decoy routing schemes such as Telex and the original Decoy Routing protocol this prevents stations from suppressing response packets from the decoy, breaking any proxied connections.

While this work identifies novel techniques that a routing capable adversary can leverage against refraction networking many of the proposed attacks were found to be significantly more difficult and less effective than originally thought. In response to the original RAD attack proposals Houmansadr et al. simulated refraction deployments and considered network links with added information about cost, bandwidth capacity, and pairing relationships [69]. They find that the alternate paths identified by the original RAD attack would require significant re-organization of existing network structures within censoring countries, with some paths increasing load by a factor of 2800. To support these routes latency in network routes would increase on average by a factor of 8. Beyond this the estimations for reduction in reachability was closer to 30% of the internet for china (as it is relatively well connected to neighboring countries) and closer to 54% and 87% for Venezuela and Syria respectively whose networks are less robust. This concept of incorporating information about existing network infrastructure has since been extended to improve the effectiveness of theoretical refraction deployments by using game theory and modeling to maximize blocking potential with minimal deployments and sets of starting rules [91, 58]. While routing attacks provide an attack surface beyond that of endpoint proxies which a routing adversary can leverage against refraction proxies, large scale attacks are believed to be too expensive to be

effective against current refraction systems.

3.3.3 Second Generation

After the initial group of refraction networking proposals multiple challenges were identified. First, tangential work in censorship circumvention was evaluating fingerprint resistance and secure transports that would prevent traffic analysis attacks that none of the first generation proposals considered. These attacks are especially strong against refraction protocols as they purport to access a specific overt site, but in truth access completely separate content. Bocovich et al. proposed Slitheen to close this gap by incorporating information about the traffic shape of decoy host connections when injecting the proxy traffic. This is done by replacing only leaf resources such as images and videos allowing the overt site to load as usual while supporting requests for censored content.

The routing attacks proposed by Schuchard et al. raised a second major issue in routing aware censors and the unique capabilities they had to identify and circumvent proxy logic at middleboxes [108]. While the studies by Nasr and Gosain responding to the RAD attacks focus on creating as much coverage as possible using tier 1 ISPs and ASes upstream from censoring countries [91, 58], networks further from censors and closer to the edge of the internet are much more difficult to route around and easier targets for refraction deployment. Nasr et al. relied on this insight in their design of a refraction station that operates only on the downstream connections of BGP routes which are much more difficult for censors to route around [93]. While this effectively addressed many of the capabilities of a routing censor, it limited the viable deployment networks. In a separate effort Ellard et al. proposed Rebound which built on the original *Decoy Routing* work removing its reliance on asymmetrical routing in both session initialization handshakes and steady state transmission [34]. In the proposed curveball protocol the client leaks the TLS premaster secret of a decoy connection to the station using sentinel traffic similar to Telex and sends continual requests to the decoy allowing the on-path station to inject response data from the covert destination into requests between the client and the overt decoy site. To do so responses from the covert

destination are embedded in HTTP URL field of client-to-decoy requests, which then mirror the data to the client. This protocol removed the reliance on symmetric routing, but in doing so added a large amount of traffic overhead on decoy hosts as well as significant limitations on bandwidth.

Finally, when approached about potential partnerships and deployments network operators were wary of placing the in-line blocking proxy appliances, which all first generation refraction proposals required, in their networks. The risk of losing service if the proxy appliance encountered an error was too high. To solve this issue TapDance, a successor to the Telex protocol, was designed to operate entirely from a passive network tap [136]. TapDance connection are established in much the same way as Telex — clients securely leak the master secret of a TLS connection with a decoy host to an on-path station. However, instead of using the station to terminate the connection to the decoy host, the TapDance client sends an incomplete HTTP request, forcing the decoy host to wait for the completion without responding. This allows the client to continue sending traffic, encrypted under the stations public key, to the decoy host which the on-path station decrypts and proxies to the covert site. Response packets from the covert site are injected into the TLS session as though it is the decoy host responding. The TapDance protocol was successful in supporting refraction networking to the standards of ISP operators and was deployed in a test configuration at a state level research network [49].

3.3.4 Current Generation

The current generation of refraction proxies continue to incorporate developments in network and censorship research. Over the last decade programmable switches have become more prevalent and new technologies like software defined networking (SDN) are being incorporated into both ISP networks and more recently, refraction. Sharma et al. proposed Siegebreaker, a refraction protocol that utilized SDN to both filter down the traffic that a refraction station is required to inspect and manage network configuration to forwarded tagged flows [112]. This outlines numerous opportunities for improving the production refraction protocols at the network layer reducing station resource requirements.

At the same time, the current generation of refraction proxies benefit from the previous generations demonstrations of refraction in theory versus refraction in practice. The TapDance deployment showed that refraction was effective at circumventing censorship, but it did not support long lived connections well. This significantly impacted the deployment as the covert addresses were primarily proxy endpoints that TapDance wrapped which typically provide clients with long lived tunnels. Because clients use an incomplete HTTP request to prevent the decoy host from responding, the decoy is required to keep the connection open and continually ingest any data the client sends while waiting for the client to complete the request. Eventually the decoy will close the connection due to connection timeout or data upload limits. On average this took around 20 seconds after which client would need to renegotiate their connection with the TapDance station. A second lesson from the TapDance deployment has been that ISPs are often geographically dispersed, and refraction systems must handle traffic at multiple network ingress points. This was a particular challenge in TapDance as the the station would attempt to multiplex sessions across multiple decoy connections, but guaranteeing that any two decoys route past the same tap is a non-trivial challenge. Instead TapDance opted for the architecture shown in Figure 3.1 which allowed detectors to perform a handshake before tunneling traffic to a central proxy node that managed sessions. The regular reconnects added significant delay and caused an identifiable sawtooth pattern in bandwidth usage. Beyond this clear fingerprint TapDance was not resistant to the traffic analysis attacks identified by Slitheen. Despite the challenges and shortcomings of the TapDance protocol, it has been running in production for over 3 years now [121].

Conjure was designed as a successor to the TapDance system and has worked to incorporate as many of the good ideas and hard learned lessons from previous refraction protocols as possible [50]. First and foremost Conjure implements the tap based architecture of the TapDance system. However, instead of parasitizing a connection to a live host Conjure allows clients to connect to the unused address space that routes past a refraction station and responds as though it is a server at that address with limited reachability. Because the default state of the endpoints in use is to never respond to traffic, Conjure limits exposure to RAD, asymmetrical routing, and probing

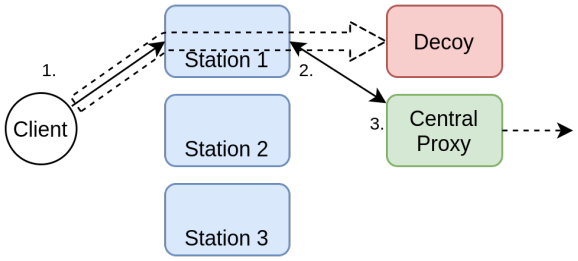

Figure 3.1: **TapDance Deployment** - To establish a TapDance connection a client first opens a TLS connection with a decoy site(1.) and securely leaks the tls secrets to a station deployed at an on-route tap. The client then prevents the station from responding by sending an incomplete HTTP request. The station pulls information out of the TLS connection(2.) and forwards it to the central proxy which establishes the connection to the covert destination on behalf of the client(3.), responding as though it is the original decoy server. The centralizing proxy allows for transparent reconnects past any tap station after the first decoy server eventually reclaims its resources by terminating the connection.

attacks.

Conjure uses a two stage process to establish a proxied connection, similar to the architecture of the Cirripede system. To effectively circumvent censors, both of stages must be non-trivial to block. The first is the *registration* in which a client generates and shares connection parameters with the station. This process is designed to be modular in order to keep clients online if any one registrar is blocked by quickly transitioning to another. Currently Conjure supports three independent registrars. The first uses a TapDance (and Telex) style tagged connection that embeds session details in an HTTPS GET request using an X-IGNORE header and securely leaking the TLS keys and registration information to a listening station as described in the original Conjure paper. The second registrar is a simple http API which can be used in a a direct configuration or composed with a third part proxy (e.g. domain fronting) intermediary, as shown in Figure 3.2. The third is a DNS based registrar that allows clients to send short (100–200 byte) messages in DNS `A` queries to the listening registrar which implements the functionality of a DNS name server to provide responses in `TXT` records. This design is based on, and operates very similarly to, the DNS-TT proxy proposed by David Fifield [42].

The second stage is *connection* during which a client connects to a Phantom address, using a "transport protocol" that wraps the clients connection. Phantoms are chosen during the registration

phase from a set of subnets pre-shared with the user in a configuration file. Again, the transport protocols are modular and swappable to make them as difficult to block as possible. Currently Conjure supports a minimal transport which prepends the data stream with a 32 byte random during connection initialization (to allow for de-duplication when phantom tuples collide), and Obfs4 where each connection generates unique key materials based on the secrets shared in the registration. The original Conjure paper proposed a third transport similar to the Slitheen protocol that would allow a client to specify a "Mask Site" that the station would mimic and embed client traffic within, though this has not yet been used in production due to implementation complexity. The transport connection transfers information from the client to the station, wrapping the protocol that is sent to the covert destination. Functionally, the client operates as though it is communicating with a host at the phantom address though in reality it is the station responding by spoofing the address in any response packets. Conjure uses two contiguous network connections to accomplish this, one connection from the client to the station(phantom) and a second from the station to the covert address with data buffered and transferred between. Conjure requires no kernel level modifications to the client, making adoption relatively simple. It is important to note that the tap design of stations necessitates that any packets sent by the client to the phantom address will be routed beyond the station — any downstream routers will attempt to deliver the packets to the phantom address.

Because connections to phantom addresses can be long lived, the multi-station coordination issues facing Conjure are different than those identified in the TapDance deployment. As clients can register past any one station, and connect past any other, as shown in Figure 3.2, the stations optimistically share registrations. Conjure uses the registration API as a centralizing node that allows stations to work together. When a station validates a registration, it shares it with the API which then publishes it to the other stations.

Conjure is currently seeing a production deployment supporting several million daily users in countries around the world, transferring an average of around 10 TB per day on their behalf. The current iteration of refraction proxies continue to learn from and incorporate the best parts of

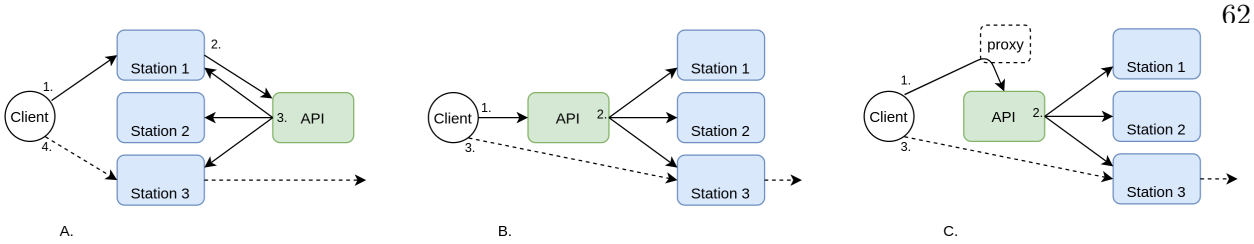

Figure 3.2: **Conjure Deployment** - With a deployment that requires stations at multiple ingress points the API acts as a centralizing node. In A. a client sends a TapDance style registration (1.) that transits past a different station than its phantom connection. To handle this, the station shares the registration with the API (2.) which then shares it over pub/sub to the other stations (3.) allowing Station 3 to handle the connection (4.). In contrast, the client in B. and C registers using the API registrar (1.) which shares the registration to all of the stations (2.) before the client attempts to connect to the phantom address (3). The client in B registers directly with the API (requiring an HTTPS connection, exposing them to IP / SNI based blocking) while the client in C. uses a 3rd party proxy to facilitate the registration request. Configuration C. is useful when the 3rd party proxy is slow or expensive, but difficult to block, as the registration is a small request and the Conjure phantom connection is long-lived and inexpensive with minimal overhead.

other schemes. With a general focus on deployability, performance, and iterable design, refraction today is reaching more users than ever before.

3.4 Design

In this section we discuss several novel designs that have been implemented to extend the functionality of refraction networking from an operational perspective. The designs in this section are motivated by both censorship strategies proposed in academic work and censorship events seen in the wild.

3.4.1 Censorship Resistance

One of the strengths of refraction networking as described in many of the previous works is the requirement that censors blocklist large sets of addresses to prevent clients from connecting. In Conjure this would require blocking all viable phantom subnets including any legitimate services hosted therein. If a censor is willing to accept the cost of blocking participating subnets they must first affirmatively identify those subnets.

3.4.1.1 Enumeration Resistance

Currently both Conjure and TapDance clients deploy a list of participating decoys and phantom subnets in the configuration built into each client. This is done because the decoys are used to initiate the connection, and the original design for Conjure simplified the control channel used for registration to only allow information transfer from client to station. This change simplified the implementation of the decoy registrar by removing the need to inject packets back to the client and limited interference with decoy sites by embedding registration information in an `X-IGNORE` header. Because stations have no way of communicating with clients before the connection phase, clients must derive the phantom address from a set of pre-shared subnets. Conjure clients choose a phantom by performing a hash-based key derivation function (HKDF) on the shared secret to select a phantom from a set of usable phantom subnets in local configuration files. Any station that receives their registration performs the same HKDF and derives the same phantom address.

However, this model of unidirectional registration is not required and actually has the relative weakness of distributing the list of all usable phantom subnets to all clients. As an alternative Conjure now supports a bidirectional registration channel where the station is able to communicate information to the client in response to their registration. In this model information like phantom address and destination port can be chosen by the station without leaking information about the larger set of available phantom addresses or the distribution of destination ports in use.

Distributing phantom addresses in this way has multiple added benefits. First it forces censors to estimate the size of the participating subnets based on addresses that they see allocated to refraction clients. This increases the probability that they either over-block causing more collateral damage, or under-block and leave phantom addresses available for use. This also gives stations more control over the algorithms that are used to assign phantom addresses to clients. Where unidirectional registrars use a pre-image resistant HKDF to prevent clients from selecting a specific phantom address, bidirectional registration channels can use more complex assignment schemes such as reputation based systems similar to those considered for Tor bridge distribution [26, 125].

This plays to the strength of refraction as large sets of proxy addresses (relative to Tor bridges) make sybil attacks less effective.

While unidirectional registration channels have their place in a broad defense against censorship, bidirectional registrars provide an exciting alternative. The API and DNS registrars provide initial support for bidirectional registration, though the decoy registrar could also be modified to support a bidirectional channel by reverting to a more TapDance like connection allowing the station to inject a small response into the client-to-decoy connection.

3.4.1.2 Passive

Beyond enumeration, one of the primary capabilities of a routing adversary is passive observation of the traffic that transits their network. Historically this tool has been used to identify proxy traffic before injecting TCP RST packets to interrupt sessions, degrading service by adding latency or reducing bandwidth, or block-listing participating endpoints.

Refraction is required to resist a number of passive threats, first of which is protocol identification. The refraction registration process can be low bandwidth or entirely out-of-band, limiting the effectiveness of protocol identification attacks. However, the phantom connection should be performant and support long lived sessions making it the primary target for protocol identification. For this reason the transport used to initiate and wrap the phantom connection must be non-trivial to distinguish from benign traffic. The transports that Conjure implements enable a number of strategies that have worked for other proxies. Currently the obfs4 and OSSH transports employ a randomizing look-like-nothing strategy. We hope to expand this support in the near future with a webrtc transport similar to the design of snowflake. This transport would blend in with other WebRTC traffic, commonly used for media and chat clients, and would allow the Conjure station to connect out to proxy clients. Beyond this Conjure could support transports that mimic other protocols such as https (as seen in Cirripede [68] and the Mask-Sites transport [50]).

3.4.1.3 Active

More aggressive censors employ active techniques to detect proxies by interacting with the connection or endpoints looking for behaviors that are unique to circumvention techniques. We consider these attacks in two separate categories, probes sent apropos of nothing and probes sent in response to a suspected proxy connection. Censors who participate as legitimate clients function in a similar manner, though we consider them more generally under the enumeration attack label.

Conjure is particularly effective against probing attacks sent out of the blue. This is because registration targets are diverse and phantom addresses are transient and particularized. To briefly consider the current registrars: the API registrar, while easily identifiable (by TLS SNI or IP address) can be accessed wrapped in any alternate proxy and requires only a single POST request. The DNS registrar similarly could be blocked by IP address, but it can be accessed recursively complicating the job plugging all routes. Further the clients original lookup can be done over encrypted DNS protocols like DoT and DoH. The decoy registrar requires that the TLS connection securely leaks the session keys using the stations public key; however, tagged traffic behaves no differently than typical requests as the information for the station is limited to an `X-IGNORE` header in a legitimate request. We can extend the decoy registrar to support bidirectional communication without compromising this probe resistance. Instead of a complete POST request the client will send an incomplete request, with connection details still embedded in an `X-IGNORE` header. The station, upon seeing this injects its response to the client as though it is the decoy responding to a complete request. The client will then close the connection inducing either a `RST` or `FINACK` from the actual decoy server. This does not require any complex connection resumption or session multiplexing, like the centralizing proxy in the TapDance implementation, because requests are small and unary allowing each station to implement this registrar independently. Because Conjure is built as a tap architecture and the client's original request prevents the decoy server from responding this registrar does not require symmetric routing. Support for other future registrars that use non-traditional channels such as internet relay chat (IRC) or email would also be resistant to this type of probing.

Probing attacks against phantoms likewise provide little to no information to censors as phantoms are drawn from the true background of the internet. If no registration exists, then the phantom address will exhibit the natural behavior of the target address — that is, live hosts will respond with protocols they speak, while unused addresses and firewalled hosts drop packets sent to them. When a registration does exist behavior depends on IP protocol version. If a censor suspects an IPv4 connection of using a refraction proxy and probes the phantom address it will again behave as the natural phantom host because refraction stations match traffic on source and phantom address as well as destination port in IPv4 (currently all transports use port 443 though support for port randomization can be added). A censor sending follow-up probes in response to a suspicious connection in IPv4 would have to take over the clients source address in order to get the station to open a TCP connection. From there all transports require that the censor demonstrate knowledge of the secret shared during registration, which they will be unable to do. TCP connections will then be timed out without preventing any fingerprintable behavior in the TCP buffering [51]. In IPv6 connections are matched on phantom address and port (again only 443 is used currently) to support clients that connect from aggressive carrier grade NAT networks or use host based privacy addresses. While a censor sending follow-up probes will be able to initiate a TCP connection if they match the phantom address and port without taking over the clients address, they will again be stopped by the transport.

In the extreme censors can inject packets into connections in a destructive manner in an attempt to validate suspicion of proxy use. This can be an injection of random data, a replay of valid data, or a deferral of client traffic while the censor preemptively injects traffic. Conjure again relies on individual transports to handle these situations. As noted by Frolov et al. simply accepting TCP connections and listening is not an uncommon response on the internet representing 3-6% of IPv4 hosts. To improve this resistance future transports could be designed to truly never respond to traffic without proof of the shared secret. Several UDP based protocols fit this category where client data is sent in the first packet, such as Quic, DTLS, as well as protocols not directly linked to TCP such as obfs4. Due to the nature of UDP replay attacks transports built on top of

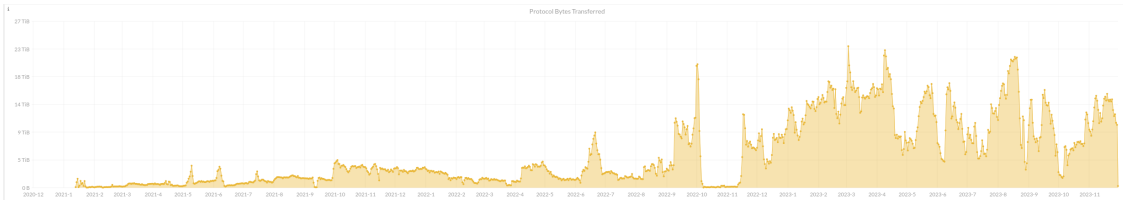

Figure 3.3: **Conjure Bytes Transferred** - Conjure has now been live for several years, however in the last year it has seen significant growth based on implementation efforts and recent events in the country where it is most heavily deployed (IR). Conjure serves several million users per day transferring above 23 TB/day at peak on their behalf.

these protocols may require some extra shared state, however these would provide a mimicking and a randomizing transport that truly look like the background of the internet to an active adversary.

3.5 Usage Trends & Analysis of a Censorship Event

The Conjure deployment is made possible by a partnership with an third party proxy client as well as several universities and research institutions. With this support Conjure has seen significant growth over the last three years. Given its tumultuous history with censorship, regularly changing blocking strategies and blocking broad sets of IPs and hosting providers suspected of hosting proxy services, Iran has been a primary focus of the refraction deployment. Figure 3.3 shows the growth in bytes transferred over that time. In aggregate Conjure system averages around 9TB of traffic per serving just an average of just under 1 million users per day, though traffic volume peaked in early March of 2023 at over 23TB/day and use connection count peaked in late August of 2023 with over 2.8 million connections per day.

By user count and bytes transferred, Conjure has seen the highest performance in the networks run by the Mobile Communication Company of Iran - MCCI (AS197207). In August of this year Conjure peaked around 2 million connections per day in MCCI alone, transferring over 16TB of traffic per day on their behalf.

By traffic share, Conjure saw the highest performance in Aria Shatel (AS31549) and Pars Online (AS16322). In these ASNs Conjure carried up to 25% of data transferred through our partner proxy on peak days and averaged nearly 10% of total bytes in peak months.

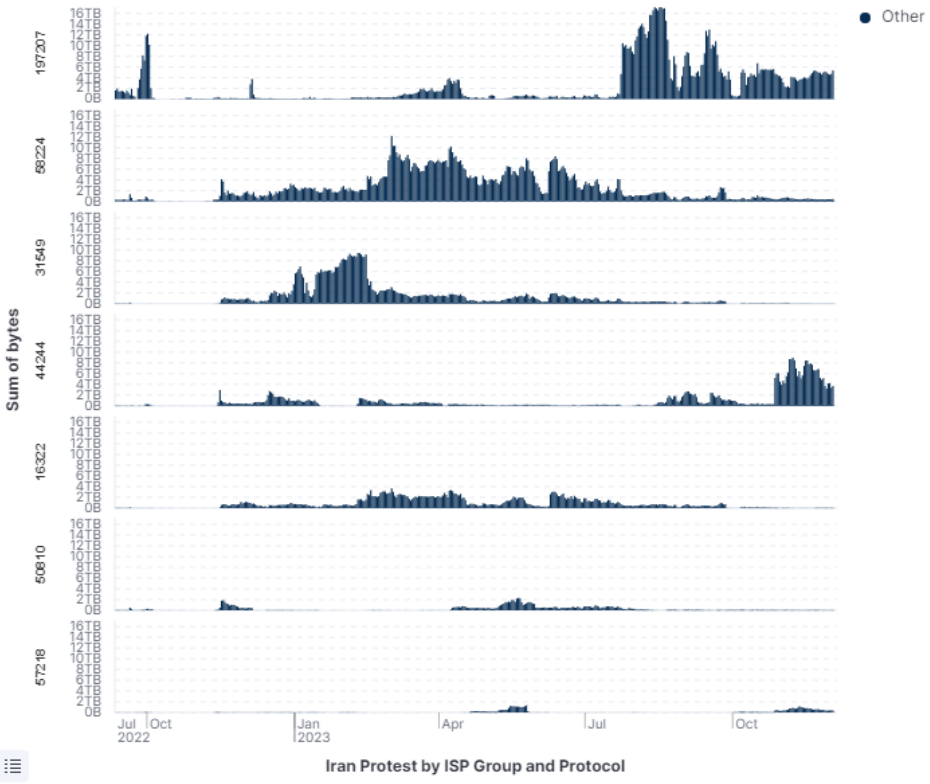

Figure 3.4: **Conjure Usage By ASN in IR** - Usage trends between the major ISPs do not align, suggesting that the censorship strategies are not centrally coordinated. Though some share overlap or may be subsidiaries of an upstream censor.

If we look at the usage of Conjure in Iran split by ASN we can see that there are several ISPs that have a high proportion of their traffic served by Conjure, however the overall usage trends between the major ISPs do not align suggesting that the censorship strategies are not centrally coordinated. This can be seen better when looking at a censorship event in Iran.

3.5.1 Active Censorship in Iran

In order to describe the behavior of the Conjure system in the face of active censorship we look specifically at the fall of 2023. It is important to note that given the nature of censorship in Iran, we are rarely able to define with certainty difference between expected baseline throughput and observed throughput during a censorship event. Within Iran, while the government may have a consistent ethos behind its censorship, the strategies that ISPs employ to enforce it are not

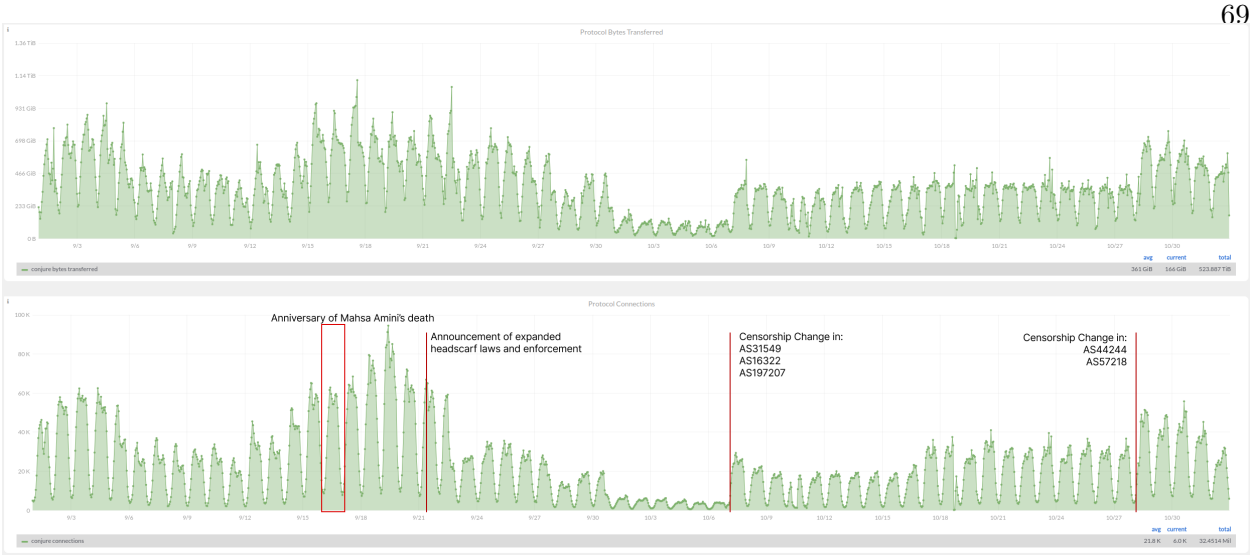

Figure 3.5: **Iran censorship event** - Ahead of the Sept. 16 anniversary of the death of Mahsa Amini the Iranian and a Sept. 21st announcement of renewed headscarf enforcement the Iranian government tightened censorship measures on the internet. This targeted direct protocols and IPs suspected to be participating in circumvention. Through this period Conjure saw increases usage as it was able to buffer the impact of the blocking, keeping users online.

consistent in time or mechanism.

With this in mind we can look at the changes in the usage and performance of Conjure and make inferences about the censorship strategies that are being employed.

In September of 2023 the we saw a change in traffic that we find to be demonstrative of a typical censorship event. Given the significance of September 16th as the anniversary of the death of Mahsa Amini, the Iranian government seemingly expanded censorship measures on the internet in anticipation of public unrest. As seen in Figure 3.5 the Conjure system saw a $3x$ increase in peak bytes transferred between September 12th and 16th. This is typical of a censorship event where one or both of the following happen:

(1) The censor blocks a large number of suspected circumvention IPs.

(2) The censor blocks traffic by protocol, typically targeting direct proxies and VPNs as well as obfuscated proxies using protocols like ssh, quick, and wireguard.

In certain ISPs the impact of a censorship change can be seen more clearly. For example

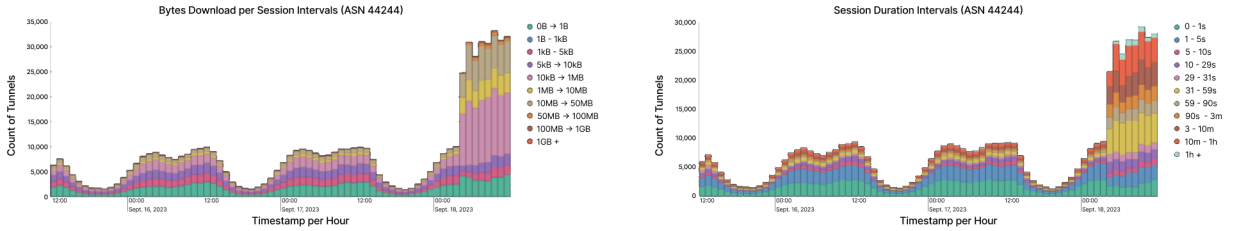

Figure 3.6: **Rising Edge of a Blocking Event** - Leading up to a blocking event in IranCell (AS44244), Conjure saw a rise in usage. This includes a shift in the distribution of session duration and bytes transferred towards towards longer lived sessions with higher overall transfer.

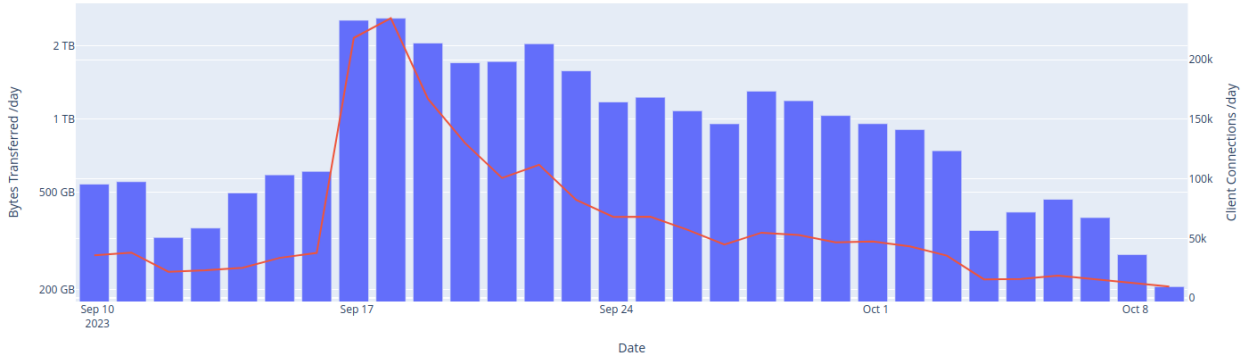

Figure 3.7: **During & after a Blocking Event** - After the initial dramatic rising edge caused by new censorship, Conjure usage slowly returns to a steady state as the proxy operator shuffles addresses and adjusts protocol availability. Conjure is able to buffer the impact of the blocking event, keeping users online until more performant options return.

Figure 3.6 shows the bytes transferred and session durations for sessions in IranCell (AS44244). We can see an almost immediate shift towards higher bytes transferred and longer sessions. Interestingly, this happened on September 18th, two days after the anniversary of Mahsa Amini's death. This is likely due to the fact that the Iranian government was preparing to make an announcement on September 21st that they would be renewing and expanding headscarf enforcement. We suspect that the expectation of unrest around this announcement could lead the government to preemptively expand censorship measures.

If we follow along as this even plays out, we see the other side of a typical censorship event from the perspective of the Conjure system. In Figure 3.7 we see that after the initial spike in bytes transferred and session counts, usage begins to taper off. This is characteristic of clients finding

solutions to the two censorship effects outlined above. In the first case, proxy administrators will shuffle addresses and hosting providers to avoid blocking, bringing fresh addresses back to clients. In the second case, clients will slowly discover other protocols that are slightly more performant than Conjure.

All in all this censorship event is characteristic of many events that we have seen in Iran. Further it demonstrates the value of the conjure system in buffering against large scale censorship ahead of significant events. In this case up to 95k users/hour stayed online because of the Conjure system.

3.6 Operating Conjure in Production

The internet itself is a complex system that involves many parties with complex relationships. Building and maintaining a refraction networking system without interfering in the existing relationships and functionalities is in itself a challenge as we build one complex system out of another. In this section we examine the performance of the Conjure system and trends we see within the outline some of the operational challenges that the current refraction deployment has met and the solutions that we have applied. These range from implementation efforts to protect interests of ISP partners, scaling services as our client grows, and adapting to the distributed structure required by ISP scale tap deployment.

3.6.1 Minimizing Destructive Impact on the Larger System

Refraction networking interacts with the surrounding networking infrastructure in different ways than any other transport protocols. This has specific implications for the way that Conjure and other refraction networking technologies can be fingerprinted and coexist with other networking entities like downstream users and services. Currently Conjure employs several strategies to prevent overwhelming impact on downstream networks.

3.6.1.1 Session Identification & Authorization

The Conjure system has two concerns when answering connections to phantom addresses. The first concern is that we not interfere with traffic destined for legitimate downstream services. The second is that we pick up for connections that the station will be able to serve (processed and ingested a valid registration) - limiting the resources that are committed to connections that we cannot serve and preventing active probing attacks attempt to identify listening phantoms.

To address these concerns in IPv4 we require that the incoming connection matches on both source and destination (phantom) address. In IPv6 we limit to destination address as there is a significantly lower likelihood of address collisions between existing downstream services and phantom addresses. However, this policy could be made more strict in both IPv4 and IPv6 by 1) matching on source in IPv6 and 2) matching on destination port number which is not currently enabled (all client traffic uses port 443) though is supported 3) matching client source port number with clients calling bind before dial to ensure predictable source port value. Any of these this would help to prevent a station from interfering with legitimate traffic unrelated to refraction networking in the situation where the destination address is in the phantom subnet range.

3.6.1.2 Phantom Host Liveness

A related challenge requires stations to prevent live hosts from being chosen as phantoms. If a downstream host that picks up / responds to TCP traffic is chosen as a phantom, when a client attempts to connect both the station and the host will attempt to respond to the TCP connection resulting in a race condition that will likely break the clients connection to the station. To prevent this Conjure attempts to probe phantom addresses upon registration before allowing the client to connect. A station is able to establish confidence that the phantom host will not interfere using probes based on large scale probing work done by Durumeric et al. [29]. This process adds latency between the registration phase and a registration being considered valid, but prevents more costly connection retries. More complex interference can also occur, such as a client becoming active after

the liveness test is performed, or the liveness test provided a false negative because the connection spoke the wrong protocol. One final consideration is that the liveness probing process will not be able to detect scanned hosts as live if they are located behind a firewall. In this situation we rely on the firewall to drop all of the unwanted ingress traffic and our address matching process to prevent the station from interfering with legitimate connections that are permitted to pass through the firewall.

The simplest solution to reduce the network interference caused by refraction connections is to use phantom subnets that are routable but truly unused. However, these subnets provide minimal value as there is no collateral cost associated with blocking them. More ideal subnets have live clients / services that provide a significant cost to Censors when blocked, though preventing interference becomes a larger concern.

3.6.2 Scalability

By design refraction networking schemes operate at scale as the middleboxes of the internet transfer large volumes of traffic continuously. This was one of the primary focuses of *"An ISP-scale deployment of TapDance"* which evaluated the ability of refraction networking to scale to the traffic rates of ISPs [49]. This work identified coordination between refraction taps deployed at disparate ingress points in the ISP network as a central challenge to deploying a successful refraction proxy.

In both refraction networking schemes that have seen production deployment this is solved using some manner of centralized coordination server. However, using centralized coordination systems has some drawbacks as deployments scale beyond individual ISPs. First in its current configuration all Conjure stations are notified about all registration, even if the phantom address associated with the registration is guaranteed to route past a different station. Second, the centralized registration sharing API is located geographically close to one ISP, however as more geographically diverse stations come online the delay from sharing over a remote API grows. Further independent ISPs may not want to directly share the refraction registrations that they serve with other networks, especially if the phantom address selected guarantees that the other network will

not be able to serve the client or worse, will potentially interfere with the connection. Finally independent networks might wish to host their own completely independent refraction network deployment while still allowing clients to interoperate.

3.6.2.1 Traffic Management

One primary source of scaling that refraction stations must address is the amount of tap traffic that is processed by the station in order to identify participating client packets. Currently the taps operate we operate process 10 - 60 Gbps traffic without saturating cores, however higher bandwidth links are becoming commonplace. One promising strategy that could be employed to scale further is to pre-filter the traffic that is mirrored to the refraction station. For Conjure specifically there is no need to operate on server response traffic which means that all traffic from server to client can be thrown out as not participating in the Conjure system. We find that a majority of the traffic that transits the tapped links is actually server-to-client traffic. The exact proportion changes by partner network, as research and academic networks seem to have a lower proportion than commodity networks. This makes a network based reduction in traffic as simple as filtering the mirror traffic with a rule like: "`not src port 443`" which will remove all server-to-client TLS traffic. Network appliances are specially designed to perform filtering operations like this at high speed.

Traffic management is also a central concern of ISP partners as there are complex legal implications associated with providing external organizations access to (in some cases unencrypted) customer traffic. One potential avenue to reduce this concern is to provide phantom addresses only from unused allocations though, as discussed before, this reduces the collateral impact to censors who broadly blocklist phantom subnets.

Figure 3.8 demonstrates the effective reduction seen when applying several potential filtering strategies to tap traffic. Filtering to `not src port 443` provides a more moderate 2–3x reduction, but maintains all flexibility for the station. Filtering to `dst port 443` only results in a consistently large reduction ratio (6–20x less traffic), but limits the flexibility of the refraction station. However,

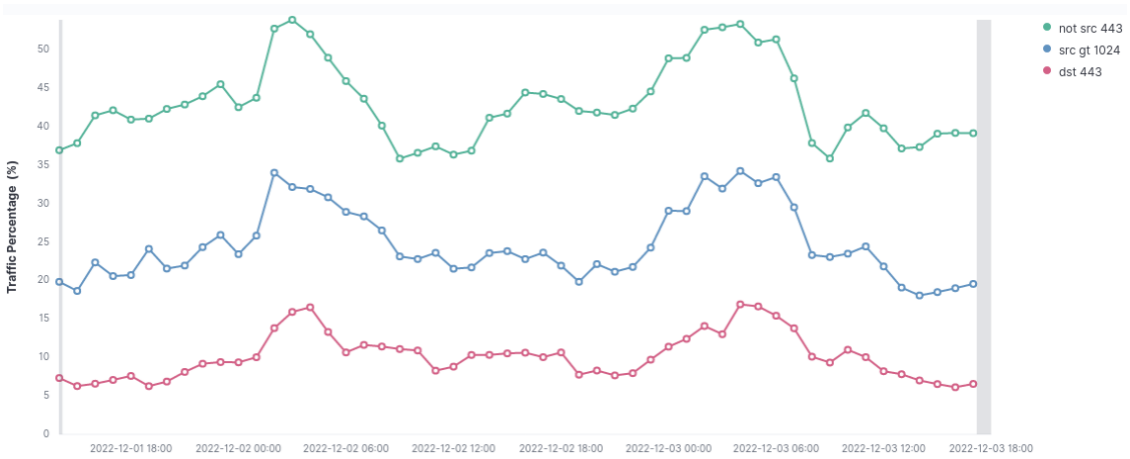

Figure 3.8: **Tap Traffic Filter Percent** - One opportunity for reducing the overhead that stations incur is to prevent taps from forwarding irrelevant traffic. Here we examine several filters and their average hourly traffic compared to the raw tap over a two day period.

this strategy has the benefit of limiting the traffic that the station is able to see to client-to-server TLS traffic, almost all of which is encrypted, potentially limiting the perceived risk for partners. Filtering out traffic with the source port in the privileged range (less than 1024) is potentially the most promising strategy, providing a middle of the road compromise allowing for flexibility in the destination port that Conjure stations can use for transports, limiting the amount of data that the station receives relating to the most common protocols, and providing a reduction of 3–5x in traffic volume.

3.6.2.2 Distributed Session Management

The distributed deployment displayed in fig. 3.2 introduces two challenges related to scaling session management in the Conjure system. This happens specifically in deployments with multiple stations at independent tap locations that share a single public key. The first challenge involves traffic that routes past more than one station. In one partner ISP deploying this configuration it is not guaranteed that a flow will only traverse a single tapped link. If a registered flow that routes past more than one station were to attempt to connect it would result in both stations attempting to service the connection - each sending a `SYNACK` to answer the client with an independent se-

quence number. The client then establishes their connection using the first `SYNACK` that it receives transmitting their `ACK` and subsequent data. To the second station it appears as though the client is using incorrect sequence/acknowledgement numbers which results in the second station tearing down the connection with a `RST`. In order to accommodate this configuration and prevent the contention in session establishment stations can provide either an allowlist of phantom subnets that will be affirmatively supported, or a blocklist of phantom subnets that the station will ignore.

This leads into the second issue with distributed session management, which is ensuring that the station responsible for answering a connection is notified of the registration in time. It is important to note that the centralizing API is not aware of the allowlists or blocklists of the various stations, and dynamic changes in routing policy could change which station is responsible for picking up an incoming connection even without a change to the station configurations. Because of this, the centralizing api notifies all stations of all incoming registrations and then stations themselves are responsible for deciding which they ingest and attempt to service. This is done by applying allowlists and blocklists early in the registration ingest pipeline to prevent stations from expending resources on connections that they will be unable to serve - for example committing a thread to perform a liveness test against a phantom address that the station cannot serve.

3.6.2.3 Federating Independent Refraction Deployments

The threat model that we propose when considering independent refraction constellations is similar to that of the Tor network [24]. Each deployment operates independently and shares registrations only to nodes that it operates via registration sharing API. This mutual distrust at the deployment scale ensures minimal leakage of client information while maintaining functionality. To allow clients to connect to any one of these independent deployments we propose an extension of the current client configuration system. This would allow a station to serve a client configuration that specifies the types of registrations that are supported, the decoys that can be used (when using the decoy registrar), the phantoms that can be used (if unidirectional registration is supported), and the stations public keys. Refraction clients are distributed with some initial configurations, but

on initial connection they can pull new or updated configurations from each independent refraction deployment. Further trusted stations can link configurations of external deployments such that clients can connect to independent deployments not included in the initial configuration set. This would allow clients to discover new deployments without direct technical involvement in updating the list of client configuration endpoints, or downloading a new APK.

Satisfying these design requirements and allowing users to form connections through independent refraction deployments will assist in growing refraction networking as a proxy technique by dispersing any risk of retaliation across independent parties, making the system as a whole more difficult to block.

3.7 Challenges & Open Questions Relating to Refraction

Refraction networking benefits from the design lessons learned from other proxy technologies, but it also experiences some open challenges unique to its own design. In this section we outline some of those unique challenges and potential jumping off points for future investigation.

3.7.1 Routing Predictability

As discussed in Section 3.3.2 effective ways to distribute refraction networking stations in order to achieve greater coverage and resist blocking have been investigated in response to routing capable censors. However, production refraction networking deployments suffer an inverse problem. The ISPs that deploy refraction stations are static and the address spaces that route past them may be different from various vantage points around the world. The simplest situation to consider here is an address block which is connected to the internet via one upstream provider. While the global internet can use that address block as phantoms, the same may not be possible for clients inside that network as they may have a more direct connection that doesn't route past the station. For clients internal to the network, every external address becomes a potential phantom address as traffic will route past an ISP tap. Extrapolating from this simple example, transit ISPs provide a unique challenge as predicting routes that transit their network is much more dependent on the

ISP peerings between the client and the participating transit ISP.

3.7.2 Key Management

Currently Conjure does not support a simple process for updating the keys that are used to connect, however large scale adoption involving multiple independent deployments would require some mechanism to achieve this. An ideal mechanism for doing so would be transparent to users allowing them to connect continuously without opening the door to censors to impersonate refraction stations. Additionally station operators may wish to deploy relatively independent keys at each station in a dependent deployment, preventing any one station that becomes compromised from effecting client traffic to other stations. One feasible solution would be to expand the dynamically delivered client configuration to include a list of keys associated with other public refraction deployments based on some consensus that prevents any one party from fabricating key information of another, similar to the design of the Tor consensus. This necessarily introduces its own design and operation issues. For now, key management remains an open challenge for a growing refraction ecosystem.

3.7.3 Quantifying Censorship in IPv6

As discussed IPv6 provides a significantly expanded opportunity to spread the phantom addresses that clients use. It also lowers the barrier for entry as publicly routable subnets are easier to acquire in IPv6. Conjure currently supports IPv6 phantom subnets and treats them as nearly identical to IPv4 from the perspective of censorship circumvention, however it is not clear that censorship in IPv6 will be directly comparable with IPv4 as the dynamics of blocking by IP or subnet change when address availability increases. For this reason it is important both for Conjure and more generally to understand how the censorship regimes in IPv6 compare the the relatively well studied systems that are applied to IPv4 traffic.

3.8 Conclusion

Refraction networking systems have gone through several iterations now from proposal, through proof-of-concept , into deployment. The feedback from previous deployments motivated the design of the conjure system which has not seen a substantial deployment itself. However, the Conjure deployment is unique in that it is now scaled to millions of daily users sending terabytes of traffic per hour introduces new and unique scaling challenges given the context of refraction networking.

In this work we cover the current design (and motivation) for the Conjure deployment as it exists today. We also cover the strategies that we have employed in scaling and adapting Conjure to accommodate the scaline usage as well as scaling deployment and requirements from partner ISPs. These efforts have driven measurement and feature integration to ensure that the impact of Conjure on non-refraction traffic and hosts is minimal, and that clients can securely connect in countries where censorship circumvention makes the biggest difference. We cover latest developments in the conjure system. Adding support for new bidirectional registrations and new out-of-band registrars gets users connected to Conjure faster and more reliably than ever. Once registered traffic analysis resistant transports create a defense in depth for securing and obfuscating client traffic.

By quantifying the impact of refraction and creating well defined strategies for managing and federating independent deployments refraction can safely scale as a part of the censorship circumvention ecosystem.

Chapter 4

Mechanizing WebAssembly for Censorship Circumvention

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 chapter, we introduce *WATER* (WebAssembly Transport Executables Runtime), a novel approach to pluggable transport design that enables applications to use a WebAssembly-based application-layer to obfuscate network traffic (e.g., TLS). 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.

Censors do not often give notice before deploying new blocking techniques, so circumvention tools must be able to adapt quickly. At the same time, the better that circumvention tools blend in with normal traffic the higher the false positive rate for censors — i.e the more benign traffic that is blocked, increasing collateral cost. *WATER* helps shrink this cycle from blocking to deployment by providing a uniform interface to rapidly transition to any protocol that remains unblocked.

4.1 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 [134, 3, 139, 124].

Because of it's 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 [100]. In response, Tor developed obfsproxy [102], which encrypts all of its traffic including protocol headers in an attempt to evade protocol fingerprinting attacks [53, 67]. While successful in the short term, this was again insufficient as in censors like China deployed **active probing** attacks to detect early versions of the protocol [131, 36], which prompted circumventors to develop probe-resistant proxies [132, 141]. Censors then found and exploited other side-channels and vulnerabilities to differentiate circumvention traffic [3, 52]. Once these problems were addressed, censors began using other features to detect and block fully-encrypted proxies such as entropy measurements [134], and circumventors responded by using prefixes that fool these measurements [47, 134].

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.

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, *WATER* (WebAssembly Transport Executables Runtime), uses WebAssembly, a binary instruction format that has runtime support across several platforms including web browsers and mobile devices. WebAssembly applications can be written in high-level languages like C or Rust, and compiled to a cross-platform binary. These binaries can make low-level system calls to make network connections,

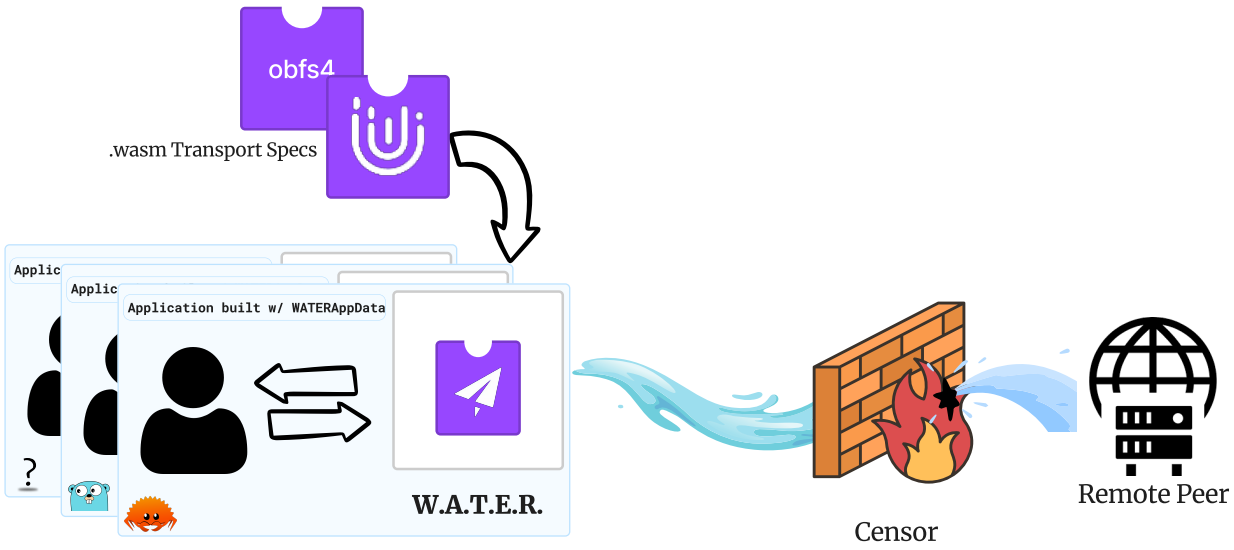

Figure 4.1: The overview of *WATER*'s role in action. With transport specs defined by `.wasm` files distributed out-of-band, *WATER* can efficiently switch between transports to use.

thanks to standards like the WebAssembly System Interface (WASI) [20]. We extend these tools and standards to make *WATER*, a library that allows circumvention tools to integrate portable circumvention techniques.

To use *WATER*, a circumvention tool integrates our *WATER* library in their client. Then, circumventors can build cross-platform *WATER* binaries that implement new circumvention techniques. These *WATER* binaries can be distributed to users over existing data channels. For instance, a technique that implements a fully-encrypted proxy could be written once, and distributed to a myriad of circumvention tools as a WATM, despite the tools using different software languages and libraries.

Because *WATER* binaries can be shared as data, they can be easier to distribute than existing software updates, which may have to take place over censored mobile app stores or through blocked CDNs. In contrast, *WATER* binaries can be distributed over any available channel.

WATER has a distinct advantage over prior approaches that provide similar flexibility, such as Proteus [122] or Pluggable Transports [60]. Because *WATER* leverages WebAssembly, new techniques can be written in one of several (and growing [127]) high-level languages. In contrast, Proteus

requires techniques be written in a bespoke domain-specific language (DSL) that limit imports of other code or libraries, and must be entirely self-contained. Meanwhile, Pluggable Transports must maintain a separate language-specific API for each supported programming language (currently Go, Java, and Swift). In contrast, *WATER* binaries can run in any WASI-compatible runtime, which currently includes support for Go, Rust, Python, C/C++, etc. [129], and can be compiled from multiple languages including Rust and Go [127], which are among the most popular programming languages used by the circumvention community. In addition, *WATER* is positioned to take advantage of 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 *WATER* binaries, and compare their flexibility and performance to existing tools.

4.2 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 [60] offers a standardized interface that different tools can integrate into. This allows circumvention tools (like 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 [73], 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 [122]. 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 [32]. 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.

4.3 Design

There are two key components in *WATER*: 1) a runtime library that is integrated into a circumvention tool to allow it to run *WATER* binaries and 2) a WebAssembly-based Transport Module (WATM) that is the interchangeable *WATER* binary implemetning a particular circumvention or transport encoding technique.

4.3.1 *WATER* Runtime Library

The runtime library is designed to be easy to integrate into circumvention tools, allowing them to run interchangeable WATM binaries that implement different circumvention strategies. The runtime library provides a WASM runtime environment for running the WATM binaries, and also presents a standard interface between the integrating circumvention client and the interchangeable WATM binary. This allows the client to make calls to the WATM binary regardless of what it is (e.g. `_water_connect()`), and for the WATM binary to be able to interface with external resources (e.g. make TCP connections, logging, etc.).

As depicted in Figure 4.3.1 — when the client calls `water_connect()` in the runtime library, the WASM module is launched, and the WATM-defined connect method is invoked. This method may choose to make a TCP connection (or multiple), which it does by calling `connect` provided

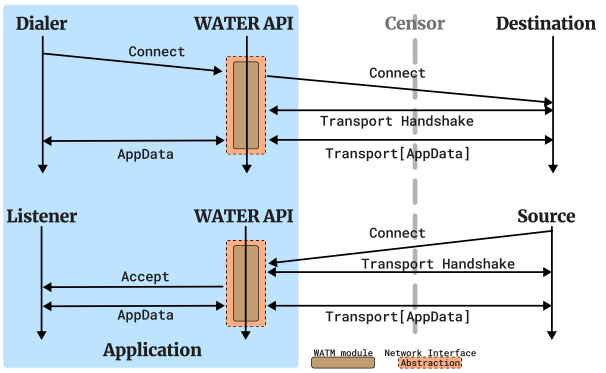

Figure 4.2: Example connection establishment flows of traditional client (*Dialer*) and server(*Listener*) each using a *WATER* transport. The dialer actively connects to a remote host upon request by caller, with the *WATER* network interface internally managing sockets and IO allowing the WATM to transforms 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.

by 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 to a WATM-defined write, which can encode it however it chooses and send it out its corresponding TCP connection. Similarly, when the client reads from the virtual socket, it does so through the runtime library and WATM-defined read function, allowing the WASM module to transform received data. In short, the WATM binary can make arbitrary transformations to the data sent by the client or received from the network in order to implement any circumvention technique.

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

4.3.2 WebAssembly Transport Module(WATM)

A WebAssembly Transport Module (WATM) is a program compiled to a WebAssembly binary that implements a specific set of functions, allowing the *WATER* runtime to interact in a consistent manner while allowing interchangeable WATMs the flexibility to apply arbitrary transformations to network traffic. For example, WATM binaries could wrap a stream in TLS (implementing TLS within the WASM binary), could add reliability layers (e.g. TurboTunnel [43], or could arbitrarily 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 binary 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 WebAssembly modules, we used it for our initial WATM binary 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*-compatible runtime.

4.3.3 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 [81]. The isolation provided is a strong defense, reducing the likelihood of adverse security impacts on the host environment. However, even with this isolation, malicious WATM binaries could still make arbitrary connections and potentially leak sensitive data from a circumvention tool that uses an arbitrary WATM binary. As with any software, it is important that we provide a path of trust, using things like code signing and verification to ensure that the WATM binary 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, in that they should be packaged and signed by trusted parties (e.g. circumvention tool developers) before being loaded into a circumvention tool.

4.4 Implementation

4.4.1 Runtime Library

We have written *WATER* runtime libraries in both Go and Rust to demonstrate the cross-platform and cross-language abilities of our approach. Each library provides a network programming interface mirroring that of the standard library for their respective programming language. To

avoid excessive duplicated work in implementing a new WebAssembly runtime and keep the design of *WATER* maximally runtime-independent, we only employed standard WebAssembly and WASI interfaces which must be implemented by every runtime library that implements the current stable standard in full [4, 117, 6, 48]. Currently, both versions of *WATER* are built with *wasmtime* [4].

In addition, we will provide starter code, examples, and detailed documentation for developers to build their tool with *WATER*. The open-source repository will be made public at a later time.

4.4.2 WebAssembly Transport Module (WATM)

Currently Rust (and C via Rust ffi) is the only language with complete support for compilation to WebAssembly + WASI. Support for wasm-wasi compile targets in other languages are a work in progress by the WASM community [107]. As support for the wasm-wasi compile target is added to more languages, it will ony become easier for circumvention developers to adapt their existing codebase to run as WATMs. For languages like golang, which is popular with circumvention developers, porting an existing library will likely be as simple as and implementing an interface allowing the *WATER* runtime to configure and launch the WATM with the appropriate role(s) from the original library (client, server, peer, etc.).

4.4.2.1 Provided Examples

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

plain.wasm implements an *identity* transform WATM that simply forwards the bytes as-received bidirectionally.

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

shadowsocks.wasm demonstrates that WATMs can implement more complex protocols, such as Shadowsocks. Rather than mimic Shadowsocks as prior work has done, *WATER* is able to build the real Shadowsocks protocol that works with an unmodified server running `shadowsocks-rust` [23] v1.17.0. To build the shadowsocks.wasm, we started with the showsocks-rust code, and

identified a 417-line file that implements the core part of Shadowsocks (i.e. encryption, decryption, and message framing). We removed all but the default features to minimize code size, and added 142(15% of total) lines of wrapping code to interface with our WATM template (more details are provided in Appendix A.3).

Reusing the original codebase allows us to easily apply updates to our shadowsocks.wasm based on upstream changes. For example, we successfully applied the exact patch [56] used by shadowsocks-rust to defend against the China's blocking of fully-encrypted protocols [134], without any changes to the patch.

4.5 Evaluation

We evaluated our *WATER* implementation, and compared latency and throughput with Proteus [122] and native network performance. The evaluation was conducted on the CloudLab testbed [28] on *c6525-25g* (16-core AMD 7302P at 3.00GHz, 128GB ECC RAM).

4.5.1 Performance Metrics

Travel through	Latency	Throughput
vanilla-SS(Baseline)	116us	2310 Mbps
***WATER*-SS-v1**	**+493us**	**2.0%**
Proteus-SS	+873us	4.8%
Raw TCP(Baseline)	26us	2210 Mbps
***WATER*-plain-v1**	**+356us**	**82.8%**
Proteus-plain	+250us	105.4%

Table 4.1: Latency & Throughput benchmark result comparing to the baseline data, with `msg_size=512`. It is worth noting that an implementation can be running slow enough to combine multiple messages into one single `send()` therefore achieving better throughput than baseline, which has `TCP_NODELAY` enabled.

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 compare the performance of each on a set of writes using buffer sizes ranging from from 1B to

4096B as 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 an identity transform for the *plain* protocol, and the shadowsocks psf implemented as part of the original paper [122] for the *shadowsocks*. We note that while the shadowsocks psf obfuscates packets using the shadowsocks format it does not actually implement the shadowsocks protocol.

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, *WATER*'s performance is highly promising.

The notable degradation in throughput for both *WATER* and Proteus when evaluating shadowsocks is in large part due to the additional cryptographic operations required. WebAssembly runtimes currently lack hardware acceleration for these operations, resulting is higher latency and lower throughput. However, support for hardware acceleration is being actively considered by the WebAssembly community [128], and we anticipate that this will significantly improve *WATER*'s performance in the future. We examine the performance of cryptographic operations in WebAssembly further in Appendix A.2.

Along with this we note that within the testbed the experiments were capable of saturating the network link resulting in over 2 Gbps for the native transports. When we consider that *WATER*-shadowsocks achieved around 50 Mbps, we believe that *WATER* is unlikely to be the bottleneck for most real-world circumvention applications. Further performance based analysis can be found in Appendix A.4.

4.6 Discussion

4.6.1 Advantages and Limitations

Maximized code reuse Beyond the transferability of the WATMs, the use of WebAssembly also enables existing tools (implemented in languages capable of compiling to WASM) to easily be converted into new WATM modules. Appendix A.3.2 investigates this further, examining the code changes we made while porting an existing circumvention tool to *WATER*.

(Temporary) WebAssembly Limitations Given the limited official support for WASI in other languages, all of our proof-of-concept WATM modules are implemented in Rust. However, we see promising trends indicating that other programming languages are making efforts to support compilation to WebAssembly using WASI standards. For example, TinyGo can be compiled into WebAssembly (though there are currently some performance drawbacks relating to the golang garbage collector). Go is widely used by the censorship circumvention community and demonstrates the potential that growing WASI support provides alongside *WATER*.

We 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 [128]. We expect that these features will only improve the potential of *WATER*.

4.6.2 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 like OONI [45], Censored Planet [115], and Ripe Atlas [113] 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.

4.7 Conclusion

By lowering the barriers of deploying circumvention techniques using *WATER*, we 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 network libraries to facilitate integration into existing circumvention tools.

Chapter 5

Conclusion

In this thesis I have investigated the design of secure systems with a specific focus on a unique resilience property. This property leverages the power that collateral cost has on the adversarial parties to make directed attempts to prevent the system from being used.

5.1 ExSpectre: Achieving Non-Deterministic Behavior using Spectre

We begin in Chapter chapter 2 in the context of hardware based secure computation.

Speculative execution is such a benefit to performance that it will never be completely removed from modern processors. However, using speculative execution we can design cooperating programs that hide their core logic from inspection / introspection. This is a demonstration that modern processors cannot simply use static/dynamic analysis to model the complete scope of the behavior that a program can exhibit.

For example, even with all of the efforts to mitigate speculative execution based attacks, type-1 speculative execution still has no fix. This is because the direct branch predictor is so fundamental to the performance of modern processors that it cannot be removed. In general existing speculative execution mitigations are designed to prevent one program from exfiltrating data from an unwilling program or portion of the system. This does not cover the case where the program itself is willing and even cooperating in an attempt to have it's behavior influenced by another.

This is the core principle of the **ExSpectre**. By leaning into the non-determinism of the speculative execution functionality that we are unwilling to give up, **ExSpectre** demonstrates that

we allow (sets of) programs to leverage and influence each other in order to hide their core logic in this non-determinism.

5.2 Refraction Networking

While hardware security and **ExSpectre** provide a relatively direct measure of the trade-off inherent in the collateral cost of system design choices, the cost is not always so tangible.

In Chapter chapter 3 we take a broader view of the effect that designing with collateral cost in mind can have on secure systems. We explore the way that Refraction Networking can be used to provide more robust and resilient censorship circumvention.

Participation in IP based networks is virtually a requirement for countries participating in the modern global economy.

It is difficult to know exactly what exists at the edge of a remote section of the Internet. Cutting of a section of the Internet is a blunt tool that can have significant economic impact. In the extreme cases countries have been known to cut off access to the external Internet entirely. However this has always been a temporary measure only employed in the most extreme cases because of the economic impact.

Traditionally censors have worked to identify individual hosts that provide circumvention services and block them. This is a game of cat and mouse from innovation to deployment to discovery to blocking of individual hosts is generally disfavorable for circumvention tools.

5.3 Water: WebAssembly Transport Design

In the context of censorship circumvention, the collateral cost of a large scale system depends not only on who is blocked, but also what is blocked. A key part of the cat-and-mouse game that is circumvention tool deployment and discovery is in identifying network protocols that can be used to ferry traffic without presenting unique characteristics. Originally this meant removing things like plaintext keywords, but as censors have become more discerning this has includes things like the ordering of the set of cipher-suites offered by TLS [3] or the entropy present in any given data

packet [134]. When a new blocking strategy targeting one of these features is deployed, it can be a significant effort to adapt existing circumvention tools to avoid detection. And each tool must be adapted individually in their native source language.

This inflexibility provides censors with a significant advantage. In Chapter chapter 4 we explore the design of a system that can be used to close this gap. Using the dynamic deployment techniques of WebAssembly, we propose a transport model that allows the on-the-wire profile that any given proxy presents to be rapidly reconfigured. This means that transports can be written once and shared to many clients across platforms and source languages. It also means that when a new blocking strategy is deployed, a variety of strategies can be deployed to get a thorough sense of what types of traffic remain unblocked.

Our proof of concept implementation of *WATER* demonstrates that it this is a viable approach to circumvention, and that the overhead of WebAssembly is reasonable for this use case (and likely to improve as the WebAssembly ecosystem continues to mature).

The theory behind this work is that the increased agility in transport design and deployment will allow circumvention traffic to adapt and blend in with whatever protocols remain unblocked – aspirationally, giving circumvention developers an edge.

—

As a whole this thesis serves to demonstrate the value of centering collateral cost when designing systems to be resistant to adversarial action. Specifically, identifying non-negotiable characteristics (e.g. performance, connectivity, functionality) and building around them such that the continued valuation and support of those characteristics ensure the functionality of your tool.

Bibliography

[1] Bitdefender antivirus technology, 2018.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Cryptographers' Track at the RSA Conference, pages 225–242. Springer, 2007.

[3] Alice, Bob, Carol, Jan Beznazwy, and Amir Houmansadr. How china detects and blocks shadowsocks. In Proceedings of the ACM Internet Measurement Conference, IMC '20, page 111–124, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Bytecode Alliance. Wasmtime, Aug 2019.

[5] arma. Research problems: Ten ways to discover tor bridges. `https://blog.torproject.org/research-problems-ten-ways-discover-tor-bridges`, 2011.

[6] Wasmer Authors. Wasmer: Run, publish & deploy any code, anywhere, Oct 2018.

[7] Dave Bakker, Dan Gohman, Lin Clark, Jiaxiao Zhou, Alex Crichton, John Edmonds, and Dan Chiarlone. WebAssembly/wasi-sockets: WASI API proposal for managing sockets. `https://github.com/WebAssembly/wasi-sockets`, 2021. Accessed on 2023-10-06.

[8] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In NDSS, 2010.

[9] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The page-fault weird machine: Lessons in instruction-less computation. In WOOT, 2013.

[10] Davide B Bartolini, Philipp Miedl, and Lothar Thiele. On the capacity of thermal covert channels in multicores. In Proceedings of the Eleventh European Conference on Computer Systems, page 24. ACM, 2016.

[11] Jan Beznazwy and Amir Houmansadr. How china detects and blocks shadowsocks. In Proceedings of the ACM Internet Measurement Conference, pages 111–124, 2020.

[12] Cecylia Bocovich and David Fifield. Snowflake. `https://snowflake.torproject.org/`.

[13] Cecylia Bocovich and Ian Goldberg. Slitheen: Perfectly imitated decoy routing through traffic replacement. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1702–1714, 2016.

[14] Sergey Bratus. What hacker research taught me. In Rss, 2009. Accessed: 2018-05-01.

[15] Sergey Bratus, Michael Locasto, Meredith Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. {USENIX; login:}, 2011.

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, 2018. USENIX Association.

[17] Gregory J Chaitin. Computing the busy beaver function. In Open Problems in Communication and Computation, pages 108–112. Springer, 1987.

[18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE attacks: Leaking enclave secrets via speculative execution. arXiv preprint arXiv:1802.09085, 2018.

[19] Richard Chirgwin. Google, aws ips blocked by russia in telegram crackdown. The Register - `https://www.theregister.com/2018/04/17/russia_blocks_google_aws_ip_addresses_to_get_telegram/`, 2018.

[20] Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web, Mar 2019.

[21] Sophia M D'Antoine. Exploiting processor side channels to enable cross VM malicious code execution. PhD thesis, Rensselaer Polytechnic Institute, 2015.

[22] ShadowSocks developers. shadowsocks-crypto. `https://github.com/shadowsocks/shadowsocks-crypto`, 2023.

[23] ShadowSocks developers. shadowsocks-rust. `https://github.com/shadowsocks/shadowsocks-rust`, 2023.

[24] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[25] Eric Doerr. Securing our approach to domain fronting within azure. `https://www.microsoft.com/security/blog/2021/03/26/securing-our-approach-to-domain-fronting-within-azure/`, 2021.

[26] Frederick Douglas, Weiyang Pan, Matthew Caesar, et al. Salmon: Robust proxy distribution for censorship circumvention. Proceedings on Privacy Enhancing Technologies, 2016(4):4–20, 2016.

[27] Arun Dunna, Ciarán O'Brien, and Phillipa Gill. Analyzing china's blocking of unpublished tor bridges. In 8th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 18), 2018.

[28] 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. The design and operation of CloudLab. In

Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, Santa Clara, CA, USA, July 2019. USENIX Association.

[29] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In 22nd {USENIX} Security Symposium ({USENIX} Security 13), pages 605–620, 2013.

[30] Maurizio Dusi, Manuel Crotti, Francesco Gringoli, and Luca Salgarelli. Tunnel hunter: Detecting application-layer tunnels with statistical fingerprinting. Computer Networks, 53(1):81–97, 2009.

[31] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In 2012 IEEE symposium on security and privacy, pages 332–346. IEEE, 2012.

[32] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network traffic obfuscation system. In 24th USENIX Security Symposium (USENIX Security 15), pages 367–382, Washington, D.C., August 2015. USENIX Association.

[33] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. ACM computing surveys (CSUR), 44(2):6, 2012.

[34] Daniel Ellard, Christine Jones, Victoria Manfredi, W Timothy Strayer, Bishal Thapa, Megan Van Welie, and Alden Jackson. Rebound: Decoy routing on asymmetric routes via error messages. In 2015 IEEE 40th Conference on Local Computer Networks (LCN), pages 91–99. IEEE, 2015.

[35] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the great firewall discovers hidden circumvention servers. In Proceedings of the 2015 Internet Measurement Conference, pages 445–458, 2015.

[36] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the great firewall discovers hidden circumvention servers. In Proceedings of the 2015 Internet Measurement Conference, IMC '15, page 445–458, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, page 5. ACM, 2015.

[38] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–13. IEEE, 2016.

[39] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):10, 2016.

[40] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In Proceedings

of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, pages 693–707. ACM, 2018.

[41] David Fifield. Domain fronting to app engine stopped working. `https://gitlab.torproject.org/legacy/trac/-/issues/25804`, 2018.

[42] David Fifield. Turbo tunnel, a good way to design censorship circumvention protocols. In 10th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 20), 2020.

[43] David Fifield. 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, August 2020.

[44] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. Proceedings on Privacy Enhancing Technologies, 2015(2):46–64, 2015.

[45] Arturo Filasto and Jacob Appelbaum. Ooni: open observatory of network interference. In FOCI, 2012.

[46] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs. Copenhagen University College of Engineering, 93:110, 2011.

[47] Vinicius Fortuna. Outlinevpn wiki: Disguise connections with a prefix to bypass protocol allowlists, 2023.

[48] Cloud Native Computing Foundation. Wasmedge, Mar 2021.

[49] Sergey Frolov, Fred Douglas, Will Scott, Allison McDonald, Benjamin VanderSloot, Rod Hynes, Adam Kruger, Michalis Kallitsis, David G Robinson, Steve Schultze, et al. An isp-scale deployment of tapdance. In 7th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 17), 2017.

[50] Sergey Frolov, Jack Wampler, Sze Chuen Tan, J Alex Halderman, Nikita Borisov, and Eric Wustrow. Conjure: Summoning proxies from unused address space. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 2215–2229, 2019.

[51] Sergey Frolov, Jack Wampler, and Eric Wustrow. Detecting probe-resistant proxies. In Network and Distributed System Security. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2020/02/23087.pdf, 2020.

[52] Sergey Frolov, Jack Wampler, and Eric Wustrow. Detecting probe-resistant proxies. In Network and Distributed System Security Symposium, 2020.

[53] Sergey Frolov and Eric Wustrow. The use of tls in censorship circumvention. 2019.

[54] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. 2018.

[55] gera and riq. Advances in format string exploiting. Phrack Magazine , 59(7), July 2001., 2001.

[56] gfw report. Shadowsocks patch for mitigating china gfw blocking of fully encrypted traffic. https://github.com/gfw-report/shadowsocks-rust/commit/d1cf917deebe1999044ca93965a602223de26be7, 2022.

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

[58] Devashish Gosain, Anshika Agarwal, Sambuddho Chakravarty, and Hrishikesh B Acharya. The devil's in the details: Placing decoy routers in the internet. In Proceedings of the 33rd Annual Computer Security Applications Conference, pages 577–589, 2017.

[59] Yashodhar Govil, Liang Wang, and Jennifer Rexford. {MIMIQ}: Masking ips with migration in {QUIC}. In 10th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 20), 2020.

[60] Pluggable Transports Working Group. Pluggable transports, 2014.

[61] Serge Guelton. Spectre is not a Bug, it is a Feature. https://blog.quarkslab.com/spectre-is-not-a-bug-it-is-a-feature.html, 2018.

[62] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In International Workshop on Recent Advances in Intrusion Detection, pages 98–115. Springer, 2008.

[63] Rolf Herken. The universal Turing machine: A half-century survey. 1992.

[64] John Holowczak and Amir Houmansadr. Cachebrowser: Bypassing chinese censorship without proxies using cached content. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 70–83, 2015.

[65] Jann Horn. Project Zero: Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018. Accessed: 2018-05-01.

[66] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In 2013 IEEE Symposium on Security and Privacy, pages 65–79. IEEE, 2013.

[67] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In 2013 IEEE Symposium on Security and Privacy, pages 65–79, San Francisco, CA, 2013. Institute of Electrical and Electronics Engineers.

[68] Amir Houmansadr, Giang TK Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In Proceedings of the 18th ACM conference on Computer and communications security, pages 187–200, 2011.

[69] Amir Houmansadr, Edmund L Wong, and Vitaly Shmatikov. No direction home: The true cost of routing around decoys. In NDSS. Citeseer, 2014.

[70] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In Security and Privacy (SP), 2012 IEEE Symposium on, pages 80–94. IEEE, 2012.

[71] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pages 263–274, 2014.

[72] Josh Karlin, Daniel Ellard, Alden W Jackson, Christine E Jones, Greg Lauer, David Mankins, and W Timothy Strayer. Decoy routing: Toward unblockable internet communication. In FOCI, 2011.

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

[74] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In Proceedings of the 27th Annual Computer Security Applications Conference, pages 403–412. ACM, 2011.

[75] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In USENIX Security Symposium, pages 287–301, 2014.

[76] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. arXiv preprint arXiv:1807.03757, 2018.

[77] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[78] Butler W Lampson. A note on the confinement problem. Communications of the ACM, 16(10):613–615, 1973.

[79] William Landi. Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS), 1(4):323–337, 1992.

[80] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In 26th USENIX Security Symposium, USENIX Security, pages 16–18, 2017.

[81] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20), pages 217–234, Virtual Event, August 2020. USENIX Association.

[82] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In International Workshop on Recent Advances in Intrusion Detection, pages 338–357. Springer, 2011.

[83] Zhen Ling, Junzhou Luo, Wei Yu, Ming Yang, and Xinwen Fu. Tor bridge discovery: extensive analysis and large-scale empirical evaluation. IEEE Transactions on Parallel and Distributed Systems, 26(7):1887–1899, 2013.

[84] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), 2018.

[85] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the security implications of speculative execution in CPUs. arXiv preprint arXiv:1801.04084, 2018.

[86] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In USENIX Security Symposium, pages 865–880, 2015.

[87] Matt Miller. Analysis and mitigation of speculative store bypass (CVE-2018-3639), May 2018.

[88] Matt Miller. Analysis and mitigation of speculative store bypass (cve-2018-3639), 2018.

[89] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 97–108, 2012.

[90] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In Security and Privacy, 2007. SP'07. IEEE Symposium on, pages 231–245. IEEE, 2007.

[91] Milad Nasr and Amir Houmansadr. Game of decoys: Optimal decoy routing through game theory. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1727–1738, 2016.

[92] Milad Nasr, Hadi Zolfaghar, Amir Houmansadr, and Amirhossein Ghafari. Massbrowser: Unblocking the censored web for the masses, by the masses. In Proceedings of the Network and Distributed System Security Symposium, NDSS, 2020.

[93] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. The waterfall of liberty: Decoy routing circumvention that resists routing attacks. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2037–2052, 2017.

[94] Jon Oberheide, Michael Bailey, and Farnam Jahanian. Polypack: an automated online packing service for optimal antivirus evasion. In Proceedings of the 3rd USENIX conference on Offensive technologies, pages 9–9. USENIX Association, 2009.

[95] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. Spectre attack against SGX enclave, 2018. Accessed: 2018-05-01.

[96] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, 1996.

[97] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In Cryptographers' Track at the RSA Conference, pages 1–20. Springer, 2006.

[98] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In Proceedings of the USENIX Workshop on Offensive Technologies (WOOT), volume 41, page 86, 2009.

[99] Colin Percival. Cache missing for fun and profit, 2005.

[100] phobos. Iran partially blocks encrypted network traffic, Feb 2012.

[101] Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. Ropinjector: Using return oriented programming for polymorphism and antivirus evasion. Blackhat USA, 2015.

[102] The Tor Project. Tor project: obfsproxy, Feb 2012.

[103] Ganesan Ramalingam. The undecidability of aliasing. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(5):1467–1471, 1994.

[104] Reethika Ramesh, Ram Sundara Raman, Matthew Bernhard, Victor Ongkowijaya, Leonid Evdokimov, Anne Edmundson, Steven Sprecher, Muhammad Ikram, and Roya Ensafi. Decentralized control: A case study of russia. In Network and Distributed Systems Security (NDSS) Symposium 2020, 2020.

[105] R. S. Richards and A. M. Brown. mov is turing-complete. Cl. Cam. Ac. Uk (2013), pages 1–4, 2013.

[106] Joanna Rutkowska. redpill... or how to detect VMM using (almost) one CPU instruction, 2004.

[107] sbinet. wasm: re-use //export mechanism for exporting identifiers within wasm modules, May 2018.

[108] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing around decoys. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 85–96, 2012.

[109] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Security and privacy (SP), 2010 IEEE symposium on, pages 317–331. IEEE, 2010.

[110] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. 2018.

[111] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security, pages 552–561. ACM, 2007.

[112] Piyush Kumar Sharma, Devashish Gosain, Himanshu Sagar, Chaitanya Kumar, Aneesh Dogra, Vinayak Naik, HB Acharya, and Sambuddho Chakravarty. Siegebreaker: An sdn based practical decoy routing system. Proceedings on Privacy Enhancing Technologies, 2020(3):243–263, 2020.

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

[114] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In NDSS, volume 16, pages 1–16, 2016.

[115] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. Censored planet: An internet-wide, longitudinal censorship observatory. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 49–66, 2020.

[116] Arne Swinnen and Alaeddine Mesbahi. One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques. BlackHat USA, 2014.

[117] Inc. Tetrate.io. wazero, May 2020.

[118] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London mathematical society, 2(1):230–265, 1937.

[119] Paul Turner. Retpoline: a software construct for preventing branch-target-injection, 2018.

[120] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In 2015 IEEE Symposium on Security and Privacy (SP), pages 659–673. IEEE, 2015.

[121] Benjamin VanderSloot, Sergey Frolov, Jack Wampler, Sze Chuen Tan, Irv Simpson, Michalis Kallitsis, J Alex Halderman, Nikita Borisov, and Eric Wustrow. Running refraction networking for real. Proceedings on Privacy Enhancing Technologies, 2020(4):321–335, 2020.

[122] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. Proteus: Programmable Protocols for Censorship Circumvention. In Free and Open Communications on the Internet (FOCI), pages 50–66, Lausanne, Switzerland, July 2023. Proceedings on Privacy Enhancing Technologies Symposium.

[123] Fish Wang and Yan Shoshitaishvili. angr - the next generation of binary analysis. In Cybersecurity Development (SecDev), 2017 IEEE, pages 8–9. IEEE, 2017.

[124] Gaukas Wang, Anonymous, Jackson Sippe, Hai Chi, and Eric Wustrow. Chasing shadows: A security analysis of the ShadowTLS proxy. In Free and Open Communications on the Internet, pages 8–13, Virtual Event, 2023. Proceedings on Privacy Enhancing Technologies Symposium.

[125] Qiyan Wang, Zi Lin, Nikita Borisov, and Nicholas Hopper. rbridge: User reputation based tor bridge distribution with privacy preservation. In NDSS, 2013.

[126] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual, pages 473–482. IEEE, 2006.

[127] WebAssembly.org. Compile a webassembly module from…, 2023.

[128] WebAssembly.org. Wasi proposals, Oct 2023.

[129] WebAssembly.org. Wasmtime: Language support, Aug 2023.

[130] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 109–120, 2012.

[131] Philipp Winter. Gfw actively probes obfs2 bridges, Mar 2013.

[132] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: A polymorphic network protocol to circumvent censorship. In Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society, WPES '13, page 213–224, New York, NY, USA, 2013. Association for Computing Machinery.

[133] Henry Wong. Measuring reorder buffer capacity, May 2013.

[134] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. How the great firewall of china detects and blocks fully encrypted traffic. In 32nd USENIX Security Symposium (USENIX Security 23), pages 2653–2670, Anaheim, CA, August 2023. USENIX Association.

[135] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In USENIX Security symposium, pages 159–173, 2012.

[136] Eric Wustrow, Colleen M Swanson, and J Alex Halderman. Tapdance: End-to-middle anticensorship without flow blocking. In 23rd {USENIX} Security Symposium ({USENIX} Security 14), pages 159–174, 2014.

[137] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J Alex Halderman. Telex: Anticensorship in the network infrastructure. In USENIX Security Symposium, page 45, 2011.

[138] Xueyang Xu, Z Morley Mao, and J Alex Halderman. Internet censorship in china: Where does the filtering occur? In International Conference on Passive and Active Network Measurement, pages 133–142. Springer, 2011.

[139] Diwen Xue, Reethika Ramesh, Arham Jain, Michalis Kallitsis, J. Alex Halderman, Jedidiah R. Crandall, and Roya Ensafi. OpenVPN is open to VPN fingerprinting. In 31st USENIX Security Symposium (USENIX Security 22), pages 483–500, Boston, MA, August 2022. USENIX Association.

[140] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In USENIX Security Symposium, pages 719–732, 2014.

[141] yawning. obfs4 - the obfourscator. `https://github.com/Yawning/obfs4`, 2021.

[142] Michal Zalewski. American fuzzy lop, 2015.

[143] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 305–316. ACM, 2012.

Appendix A

WATER Supplemental Materials

A.1 Extending `wasmtime` C API binding

WebAssembly System Interface Preview 1 (wasip1) was solidified without official support for network sockets and *wasi-sockets* proposal is only in Phase 2 by the time this paper was written [7] and still has a long way to go before it can make it to a preview snapshot of standard WASI APIs. A number of popular WASI runtime software libraries including *wazero* [117] and *wasmedge* [48] have implemented their own sockets support in different ways that are runtime-specific and non-standard. To make *WATER* completely WASI runtime-independent and cross-platform, we decided to keep from features that are only supported or implemented on specific runtime implementations and stick to the latest official snapshot, wasip1.

To support basic socket operations such as `dial()`, `listen()`, `read()`, and `write()`, we implemented our own WebAssembly Transport Module (WATM) API which was described in section 3.4. And the two implementations in different programming languages for the *water* runtime library provided by us are both based on *wasmtime* [4], a WASI runtime developed and maintained by bytecodealliance, who is one of the main organization supporting the WebAssembly standardization. Although wasmtime libraries have been distributed in multiple programming languages including Rust, Go, and Python, the core functionalities were written in Rust natively and compiled into C API to be linked by implementations in other programming languages. However, the C API is falling behind on schedule and has a few critical interfaces missing.

In order to provide *water* concurrently in multiple programming languages, we decided to

extend the current C API of wasmtime by adding the missing interface/functionalities. Specifically, we added a set of C APIs that are directly related to file descriptors and file I/O operations achieved via `wasmtime_wasi::WasiCtx`, a struct which is currently not accessible through the C API. Unfortunately, this functionality isn't prioritized by the authors of wasmtime and didn't receive enough attention from them when we tried to contribute back into the main repositories of wasmtime. Despite our continuing efforts in trying to merge the changes into the main branch, as an temporary solution, we choose to maintain public fork of repositories `wasmtime` and `wasmtime-go` until our changes gets accepted and merged into the original repositories. Currently the modified forks are accessible and available to general public.

The authors of this paper are willing to provide the forked repositories as well as the patch applied to the original repositories as a part of the artifact, should the artifacts of this research work be requested for inspection.

A.2 Crypto Performance of WASM

WASM is a new technology is still in its early stage of development and running in VM, therefore, it is not surprising that it doesn't have support for hardware acceleration for cryptographic operations(e.g. SIMD). However, it is still important to evaluate the performance of WASM in terms of crypto operations. We conducted a performance tests to evaluate the performance of WASM in terms of crypto operations. See table A.1 for the results.

Configuration	Native	WASM
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 A.1: Crypto Performance - MacbookPro

A.3 Implementing `shadowsocks.wasm`

We have been developing two distinct versions of `shadowsocks.wasm` and a patched version of *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. The latter version, implemented with v1_preview, is optimized for real-world applications, capable of managing network traffic and configurations. This includes server relay and bypass connections to whitelisted IPs.

A.3.1 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* [22]. 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 align with the latest advancements in WASI.

A.3.2 Porting from shadowsocks-rust

vanilla_SS **(32%)**	*WATER* _SS **(85%)**
2548-2687	810-922
168-242	80-154
...	...
1263-1266	493-496

Table A.2: shadowsocks implementation code comparison

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 A.2 based on comparing the implementation of core logic in *shadowsocks-rust* and *WATER*-SS implementation, shows the code change only **85% - 785 / 927** lines matched where only **142** lines of glue code for integrating all the points we mentioned above.

A.3.3 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 [134]. This particular implementation, designed to mitigate the blocking of shadowsocks by the GFW, was proposed by gfw-report [56] and discussed in detail on Net4People [57]. Our v1_preview version of shadowsocks successfully incorporates this patch without necessitating any additional modifications.

Table A.3 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 linked here and the *WATER*-shadowsocks Patch Commit linked here, 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%)**
8-196	6-194
198-319	195-316
1-5	1-5

Table A.3: MOSS check of `diff` on patching official v.s. *WATER*

A.4 More Benchmark Results

The subsequent sections present a detailed Table A.4 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 plain mode, while vanilla shadowsocks-rust is used as the baseline for the Shadowsocks comparisons in this figure A.1 comparing latency and throughput for shadowsocks to draw a clearer picture of the performance of *WATER* and finding the optimal packet size for *WATER* to balance the latency and throughput.

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 A.4: Plain-Relay latency/throughput - CloudLab topology

A.4.1 on Apple Macbook Pro 2021

We conducted more real-world 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 A.5, showcasing performance metrics obtained using `iperf3` to connect from Michigan to a server in San Francisco.

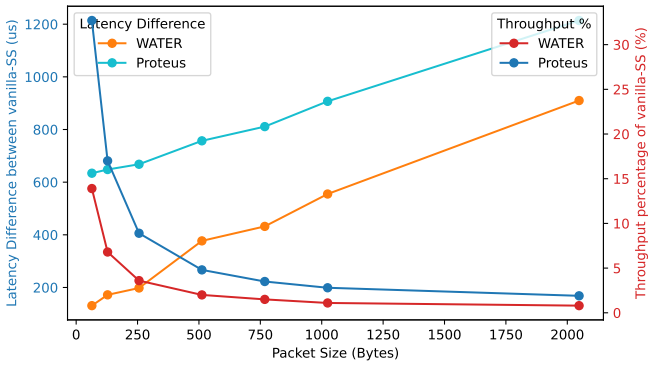

Figure A.1: Latency & Throughput Comparison with Vanilla-SS at Different Packet Sizes

Travel through	iperf3 - 10s	iperf3 - 600s
vanilla-SS	415 / 411	418 / 418
***WATER*-SS-v1**	**56.5 / 56.5**	**56 / 56**
WATER-SS-v0	50.0 / 48.3	50.4 / 50.3
Proteus-SS	96.6 / 83.5	68 / 67.8

Table A.5: Sender / Receiver (Mb/s) - MacbookPro