

Raising the Bar on Lowering Barriers: Improving Ease of Research and Development Contributions to Privacy Enhancing Technologies

by

Justin Tracey

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Justin Tracey 2024

Some rights reserved.



Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Daniel S. Roche
Professor, Computer Science Department
United States Naval Academy

Supervisor: Ian Goldberg
Professor, Cheriton School of Computer Science
University of Waterloo

Internal Members: Florian Kerschbaum
Professor, Cheriton School of Computer Science
University of Waterloo

Mei Nagappan
Associate Professor, Cheriton School of Computer Science
University of Waterloo

Internal-External Member: Douglas Stebila
Associate Professor, Department of Combinatorics & Optimization
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Statement of Contributions

All work in this thesis was done under the supervision of my supervisor Ian Goldberg. The contents of Chapter 3 are adapted from a paper coauthored with Rob Jansen and Ian Goldberg [JTG21]. Chapter 4 is coauthored with Ian Goldberg, with the intent to publish a modified form on its own. The contents of Chapter 5 are adapted from a paper coauthored with Ian Goldberg [TG23].

Abstract

As daily life becomes increasingly subject to surveillance economies and surveillance states, so do privacy enhancing technologies (PETs) become increasingly important to those who wish to live unmediated by these mass data collection practices. While improvements to the design and implementation of PETs have allowed more people than ever to take advantage of these tools, the research and development practices surrounding PETs require the work of domain experts—with their own biases, values, motivations, and awareness—doing their best to accommodate the needs and desires of users. As a result, an inevitable gap forms between what is built by those who have the skills and resources to develop these technologies, and what is needed by those who can only use what is available.

In this thesis, we examine techniques for lowering barriers to entry for research and development contributions to PETs, so that users who wish to make such contributions are more readily able to do so. Towards this end, we use as a concrete example the Tor project, and how it interfaces with current research and development practices to demonstrate three methods of lowering such barriers: (i) a set of tools and techniques for conducting statistically sound Tor experiments, (ii) an analysis of the viability of Tor as a means of providing simple metadata protection in messaging apps, and (iii) an investigation into the effect on new contributors when porting a codebase written in a programming language without memory safety to the memory-safe Rust language, as Tor is doing. We find that, using these methods, the barriers to entry can be reduced, but that considerable future work still remains in this vein.

Acknowledgments

I would like to thank, first and foremost, my supervisor Ian Goldberg. My decision to dedicate this part of my life towards research in privacy enhancing technologies was in no small part due to his work in this field, and I am eternally grateful for the ability to work with him. The time, feedback, and patience he has given me have undoubtedly made me a better researcher.

I would also like to thank my thesis committee: Florian Kerschbaum, Mei Nagappan, Douglas Stebila, and Daniel S. Roche for their feedback and time.

To Anne, Jim, Helen, and Stella, I thank for your unconditional love and support, even when we could rarely see each other. It may have taken some time, but I can now say I have upheld my promise to finish college.

Of course, this thesis would have been impossible without the friendships I made at the CrySP lab. Whether in person or on irc, the countless vortices, reading groups, D&D sessions, and adventures helped make me who I am now. I am certain I will fondly remember this time because of all of you.

Chapter 3 benefited from the feedback of our shepherd, Yixin Sun, and the anonymous reviewers. That work was partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF) under award CNS-1925497. Chapter 3 and Chapter 5 were partially supported by the National Sciences and Engineering Research Council of Canada (NSERC) under award CRDPJ-534381. Chapter 4 and Chapter 5 were partially supported by NSERC under awards RGPIN-03858 and RGPIN-2023-03260.

This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. Some of the Tor network experiments and platform development of Chapter 3 were done using the CrySP RIPPLE Facility, as were all presented Shadow experiments in Chapter 4.

Dedication

To the countless unnamed.

Table of Contents

Examining Committee Membership	ii
Author’s Declaration	iii
Statement of Contributions	iv
Abstract	v
Acknowledgments	vi
Dedication	vii
List of Figures	xii
List of Tables	xiv
1 Introduction	1
2 Background on Tor	5
2.1 Onion Services	7
2.2 Threat Model	8
2.3 Terminology	9

3	Conducting Statistically Sound Tor Experiments	10
3.1	Background and Related Work	13
3.1.1	Tor Experimentation Tools	13
3.1.2	Tor Modeling	14
3.1.3	Tor Performance Studies	14
3.2	Ontology	15
3.3	Models for Tor Experimentation	18
3.3.1	Internet Model	18
3.3.2	Tor Network Model	19
3.4	Tor Experimentation Platform	24
3.4.1	Shadow Background	24
3.4.2	Shadow Improvements	24
3.4.3	Evaluation	25
3.5	On the Statistical Significance of Results	29
3.5.1	Methodology	30
3.5.2	Discussion	33
3.6	Case Study: Tor Usage and Performance	36
3.6.1	Motivation and Overview	36
3.6.2	Experiment Setup	37
3.6.3	Results	40
3.7	Conclusion	43
4	Using Tor to Protect Metadata in Messengers	45
4.1	End-to-End Encrypted Messengers	46
4.2	Solutions to Protect Metadata	47
4.2.1	Ignore Existence	47
4.2.2	Deflect Responsibility	48
4.2.3	Integrate Solutions	50

4.3	Analysis of Metadata Protection	51
4.3.1	Methodology	51
4.3.2	WhatsApp	52
4.3.3	Signal	53
4.3.4	Tor-based apps	58
4.4	Performance Analyses of Messengers and Tor	59
4.4.1	Client Model	59
4.4.2	Data Sources	63
4.4.3	Implementation	64
4.4.4	Results	66
4.4.5	Effects of Tor on Messengers	67
4.4.6	Effects of Messengers on Tor	69
4.4.7	Discussion	70
4.5	Conclusion	71
5	The Impact of Rust on First-Time Code Contributors	73
5.1	Background and Related Work	75
5.1.1	Identifying Vulnerability Fixes	75
5.1.2	Correlating Vulnerabilities with Contributor Experience	77
5.1.3	Oxidation	77
5.2	Methodology	79
5.2.1	Data Extraction: Original Code	79
5.2.2	Data Extraction: Rust Equivalents	82
5.2.3	Examining Experience	82
5.3	Results	85
5.3.1	Identified Vulnerabilities	85
5.3.2	Learning Parameters	89
5.3.3	Analysis	89
5.3.4	The Prevalence of New Contributors	92
5.4	Conclusion	93

6 Conclusion	95
References	97
APPENDICES	117
A Additional Performance Results	118
B Rust Vulnerabilities in Oxidation	121
Glossary	124

List of Figures

1.1	Research questions.	4
2.1	The high-level design of Tor.	6
2.2	The high-level design of onion services.	7
3.1	The rate of Tor relay churn over all network consensuses from January 2019. . .	25
3.2	A comparison of simulated results with Tor metrics.	27
3.3	Results from 3 simulations at network scale $s = 1.0$	28
3.4	A synthetic example of estimating the cumulative distribution of a random variable X	32
3.5	The width of the 95% CI when sampling a number of networks.	34
3.6	Sampling error introduced by Shadow is much less significant than error introduced by Tor network sampling	35
3.7	Time to last byte of 1 MiB downloads from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale.	39
3.8	The download error rate for downloads of all sizes from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale.	42
4.1	The connections and protocols used in Signal, along with notable data provided by the client at each layer.	54
4.2	State diagram for a user’s conversation model.	61
4.3	The impact of Tor on messenger behavior.	67
4.4	The cumulative goodput of all relays.	69

4.5	The impact of Tor-enabled messengers on non-messenger Tor performance. . . .	70
5.1	Contributor experience vs. the number of vulnerabilities introduced.	90
5.2	The C++ and Rust learning curves.	90
5.3	The experience of the examined C++ and Rust Oxidation projects.	93
A.1	Time to first byte in seconds of all 50 KiB, 1 MiB, and 5 MiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s	119
A.2	Time to last byte in seconds of 50 KiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s	119
A.3	Time to last byte in seconds of 5 MiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s	119
A.4	Tor network goodput in Gbit/s: for each second in the simulation, the sum of the goodput from all Tor relays.	120

List of Tables

3.1	Classification of the experimentation properties in our Tor ontology into Independent and Dependent variables, organized by element.	16
3.2	Description of properties from the ontology.	17
3.3	Statistics computed during the <i>staging</i> phase.	19
3.4	Scalability improvements over the prior state of the art	29
3.5	Symbols used to describe our statistical methodology.	31
3.6	Tor usage and performance experiments in Shadow	38
3.7	Network composition in each simulation	39
4.1	Entropy at which a user becomes unique (in expectation).	55
4.2	Distributions and probabilities in the model.	60
4.3	Shadow experiment configurations.	66
5.1	Oxidation projects.	78
5.2	Sample size, vulnerability issues, and valid vulnerable commits for C++ and Rust projects.	86

Chapter 1

Introduction

As any technology transitions into an increasingly central role in our lives, understanding the specific properties and features of that technology becomes increasingly important. While there are toys and gadgets that are merely toys and gadgets, when a particular technology is used as our primary means of informing ourselves, entertaining ourselves, or mediating our social interactions, it elevates the importance of self-determination in how that technology behaves. One aspect of this self-determination, with particular regards to digital technologies, is the matter of privacy.

While “privacy” is a broad notion with many meanings [Sol07], in this context, privacy can be thought of as individuals’ autonomy over their data. To facilitate this autonomy, technologists and researchers have created various privacy enhancing technologies (PETs). PETs can be used in conjunction with other technologies to make those technologies more private, or as independent tools providing some privacy-related service. Some examples of these technologies include Tor, which is used to anonymize internet connections; the associated Tor Browser, which provides web content over Tor while minimizing web-based privacy risks to the user; and end-to-end encrypted (e2ee) messaging, which is used to exchange information between people in a way that only the end users’ devices have access to that information.

The development of these PETs can be thought of as having two historical phases. In the early days of PETs, research and development focused on creating tools that could be used to address the threats modeled by researchers, and coupling that with education among the general public on proper use and understanding of these tools. This era could be called the “cypherpunk” era of PETs, and is best exemplified by PGP—a tool for sending and receiving e2ee email, and the associated Web of Trust for managing chains of trust in public key pairs. Despite decades of concerted effort from dedicated volunteers and professionals, PGP and the Web of Trust never saw widespread adoption for e2ee email. Usability hurdles like manually managing keys and understanding how

to threat model trust in other users’ keys proved effectively impossible to overcome for a wide audience [WT99]. This era has been critically described as cypherpunks making tools they could use, then trying to teach everyone else how to be like cypherpunks [Mar15].

The next phase of PETs development was therefore a direct contraposition to the cypherpunk era: if requiring understanding how these tools work prevented the widespread use of PETs, then widespread use of PETs requires making tools that involve no understanding for their use. As PGP is the archetype of the cypherpunk era, then so the e2ee messenger app Signal (described in more detail in Chapter 4) is the archetype of this era in PETs development. While the underlying cryptography of Signal did push the state of the art forward with improved security over PGP, far more of its impact came in the form of its “Development Ideology”, which includes as its first four points [Mar14]:



1. The answer is not more options.
2. The user doesn’t know what a key is.
3. There are no power users.
4. If it’s “like PGP,” it’s wrong.

SIGNAL DEVELOPMENT IDEOLOGY

These four points have remained unchanged since their inception ten years ago as of this writing. Unlike PGP, Signal does not expose language around keys or threat models to users. Instead, users get the simple, cohesive experience of a fully featured messenger app, with the additional security it provides abstracted away. This ideology saw success, with Signal seeing 40 million monthly active users [Cur24], and WhatsApp, which eventually adopted much of Signal’s protocols, having over 2 billion monthly active users [Dix24]. For comparison, the most popular PGP key servers have never had more than 500,000 email addresses [key24] or 8 million individual keys [Ubu24] (only a fraction of which are valid and actively used), while Firefox Mobile, the 7th most popular mobile web browser [Sta24b], has somewhere over 100 million installs [Pla24].

However, with increased adoption of the Signal Ideology, there has not only been a shift away from the *necessity* of understanding how these tools behave and function, but also a shift away from the *feasibility* of users influencing how these tools behave and function. Be it from factors that are explicit (e.g., the opaque nature of platforms like WhatsApp and the companies that run them), implicit (e.g., funding models biasing the kinds of problems that take priority), or somewhere in between (e.g., the sentiment that these tools and systems are so fragile that

their maintenance is best left to domain experts), users who experience shortcomings in these tools outside the forefront of our minds can find themselves without recourse to address these shortcomings [EHM17]. As such, the research and development of these PETs can fail to adhere to the same principles of autonomy they aim to provide.

It is unlikely the need for experts in PETs will be eliminated in the near future. Nevertheless, lowering barriers of entry and reducing the required work involved with research and development of PETs is important, not only for the general benefit of a larger pool of effort and contributions available, but as a means of alleviating some of the implicit power imbalances involved in said research and development.

In this PhD dissertation, we establish the following thesis statement:

By examining how a particular privacy enhancing technology (PET) is impacted by, and itself influences, PET research and development practices, we can establish techniques or observations to improve safety and/or lower barriers of entry to PET research and development.

In this work, we establish this by focusing on the Tor privacy enhancing technology. We focus on four research questions, given in Figure 1.1. These questions do not fully enumerate the contributions made to improving safety or lowering barriers of entry in this work, but serve as a concise framing—e.g., our work on network simulations has proven useful not only for researchers, but also developers wishing to perform testing on (non-research oriented) code changes made to Tor. The research answering these questions is presented in three chapters.

First, in Chapter 3, we look at Tor network experimentation. The network simulator Shadow has been the most popular method of conducting safe, controlled Tor network experiments for some time. However, without a foundation in a statistically sound process of sampling and presenting data, researchers are left with ad hoc and potentially faulty research practices. To address this, we created tools and a methodology to allow researchers to confidently create safe, statistically sound Tor network experiments.

In Chapter 4, we examine the influence of Tor on the e2ee messenger space. Tor is frequently cited by researchers and developers as a way to protect network-level metadata in these messengers. We establish the extent to which it is viable to use Tor as a drop-in solution to protecting metadata in e2ee messengers, both in terms of the protection Tor provides, and the impact it has on users of the messenger and Tor. In the process, we make use of the tools and methodology established in Chapter 3.

Finally, in Chapter 5, we consider the Tor Project’s choice of rewriting Tor’s core C code in Rust. By examining other projects that have done the same, we found that doing so can make it safer, and therefore easier, for new contributors to get involved with adding code.

	Practices that influence Tor	Practices that Tor influences
Research	Tor network simulations allow researchers to conduct experiments in controlled environments. How can we help researchers ensure these experiments are statistically sound, and produce meaningful results? (Chapter 3)	Tor is commonly cited as a way to let e2ee messengers protect network-level metadata. How effective is this design in practice? (Chapter 4)
Development	Tor is in the process of being rewritten from C to Rust. How does this affect the safety and frequency of code contributions from new contributors? (Chapter 5)	Tor provides tools for anonymous communication. Do these tools let e2ee messenger developers use Tor to simply abstract away metadata concerns, without impacting overall user experience? (Chapter 4)

Figure 1.1: In this thesis, we investigate practices that *influence* Tor, and practices that Tor *influences*, with regards to privacy enhancing technology *research* and *development*. This in turn motivates associated research questions, which we address in this thesis, on improving safety and lowering barriers of entry for said research and development.

In summary: By examining how the research practice of network experimentation impacts Tor, we can lower the barrier of entry for statistically sound Tor network research (Chapter 3). By examining how the research and development practice of protecting metadata in e2ee messengers is influenced by Tor, we can improve safety and lower barriers of entry for researching and developing e2ee messengers that effectively protect metadata (Chapter 4). By examining how the development practice of porting C/C++ code to Rust could impact Tor, we can improve safety and lower barriers of entry for making PETs code contributions (Chapter 5). Collectively, this analysis demonstrates the viability of using the *interactions* of a PET with research and development practices, rather than the PET itself, to identify ways in which PET research and development can be made safer and with lower barriers of entry.

Chapter 2

Background on Tor

Before proceeding to describe our contributions in detail, in this chapter, we provide background generally useful going forward—namely, background into the operation of Tor.¹ Tor is a form of anonymizing proxy, which uses a technique known as “Onion Routing” [DMS04; GRS99]. It aims to allow access to the internet (specifically, as TCP streams) such that no single entity, aside from the user’s device, knows both the source and the destination of traffic simultaneously. In its most common configuration, it achieves this goal by routing traffic through three randomly selected volunteer machines, known as *relays*, or onion routers. The traffic sent from the client is first wrapped in three layers of nested encryption (on top of any encryption provided at the application layer, such as a TLS session with a website), with the outermost layer encrypted to the first hop, the next layer encrypted to the second hop, and the innermost layer encrypted to the final hop. Return traffic from the server is sent to the last hop, which encrypts it and sends it to the second hop, which encrypts it and sends it to the first, which encrypts it and sends it to the client, which can decrypt it. See Figure 2.1 for a visual representation of this configuration.

Each time a Tor client selects the three relays to route through, it is constructing what is known as a circuit. A client may construct any number of circuits, each of which can be used to connect any number of streams to any number of destinations. Each relay in a circuit is chosen with probability proportional to its bandwidth, subject to certain flags that indicate the type of relay.² The first relay in the circuit is known as a guard relay. To be a guard relay, the relay must be stable over a long enough period of time. Clients choose a small pool (currently configured to a default size of 2) of guard relays when first installed, and rarely update this pool after. Each

¹More particular background is present in each respective chapter.

²Other constraints, generally for preventing related relays from being used more than once in the same circuit, are also employed. These additional constraints will not be relevant for this research.

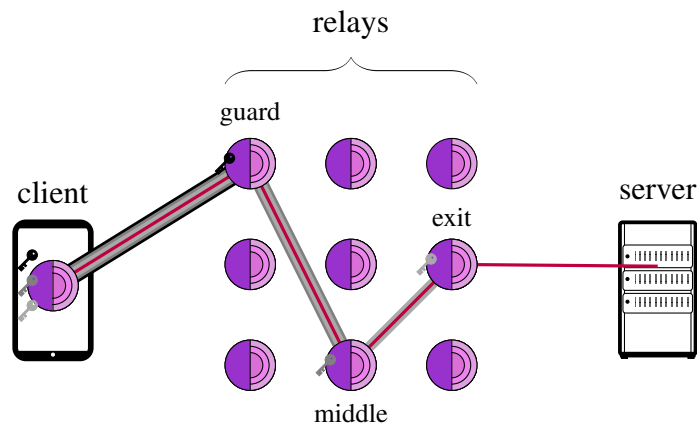


Figure 2.1: The high-level design of Tor. Tor protects client anonymity by nesting layers of encryption over three proxy hops, known as relays, or onion routers: a guard relay, followed by a middle relay, then an exit relay. Nodes running Tor are here represented as onions.

circuit selects its guard relay randomly from this pool. The last relay in the circuit, just before exiting to the rest of the internet, is referred to as an exit relay. Unlike guard relays, the status of being an exit relay is chosen by the relay operator. Most relay operators choose not to allow exit traffic, as any activity performed on that circuit will, from the perspective of the destination, appear to originate from the exit relay—a property many server operators, law enforcement, and courts would need explained to them. Because of this, exit relays make up only a fraction of the Tor network, both in count and in provided bandwidth. The second relay in the circuit is called a middle relay, and has the fewest restrictions. A relay can serve multiple roles, so long as it is not in the same circuit; e.g., exit relays can also function as middle relays, and many guard relays are also exit relays.³

In order to select these relays appropriately, clients download a file known as the network consensus, containing all relays, their weights, flags, and other useful information. This network consensus file is constructed once an hour by a small set of relays (8 as of time of writing), trusted not to maliciously collude, called directory authorities. Some information that clients do not need frequent updates for, such as the relay’s public keys, are stored in separate files called server descriptors.

³Though in practice, the limited bandwidth available for exit relays means they are rarely available for actual use as guard relays.

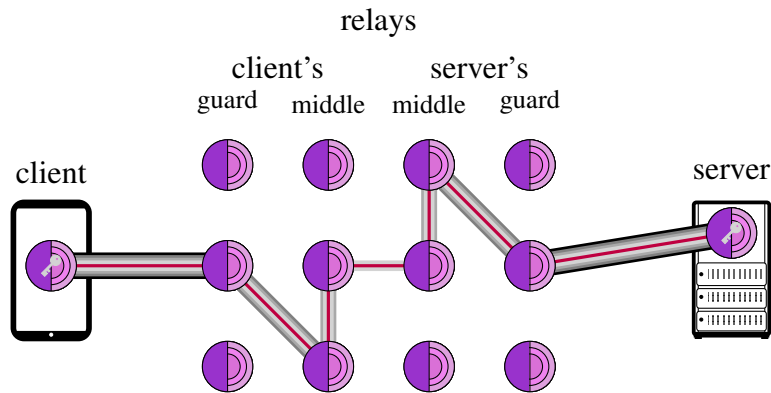


Figure 2.2: The high-level design of onion services. The client and onion service each select three relays (a guard and two middle relays) to route traffic to each other, never leaving the Tor network, and never transmitting plaintext.

2.1 Onion Services

In addition to the typical Tor usage described above, there is another primary use of Tor: onion services. Originally conceived of as a means of hiding the location of a server from the user, in recent years, onion services have taken on a more general role.

At a high level, onion services work using a public key, plus some additional data, base32 encoded into an identifier that, when combined with the reserved `.onion` top-level domain [AM15], functions as a domain name for the service, called an *onion address*.⁴ A client can use the information in the onion address to look up how to perform a handshake with the onion service, ultimately constructing a circuit with three relays chosen by the client, and three chosen by the onion service, plus an additional inner layer of encryption added to tunnel between the client and server endpoints. Thus, the traffic never leaves the Tor network, and no entity, including the client (and server), ever observes the source and destination simultaneously. Once this connection is established, the application layer can transparently use TCP streams as usual.

As previously stated, the initial motivation for onion services was that a server can operate without revealing its IP address (indeed, the original name of onion services was “hidden services”). In addition, though, onion services were found to have other uses.

For one, because their public keys are encoded into their onion address, onion services are self-authenticating—simply knowing the identifier of an onion service is sufficient to authenticate

⁴E.g., `nytimesn7cgmftshazwhfgzm37qxb44r64ytbb2dj3x62d21ljsciiyd.onion` (this link will only work with Tor).

the connection. This can provide assurances that are in some ways stronger than that of other authentication mechanisms that sign existing identifiers, such as the Certificate Authority PKI system used in TLS to authenticate the identity of a website serving content on a particular domain, or the Web of Trust used by PGP to authenticate the identity of senders and recipients of emails. The major disadvantage of this self-authentication is a lack of human-readable identifiers.

Another reason that onion services have been used in practice is they provide a ready means of “NAT punching”—being able to serve content from within a network that does not provide a static, external IP, or does not allow opening TCP ports in a firewall. For example, OnionShare⁵ allows users to share a file over the internet without needing to first upload the file to a server, which in turn allows sharing files of arbitrary size, subject only to the user’s storage space and the bandwidth of the Tor circuits used. (It also benefits from the aforementioned self-authentication, assuming there exists a trusted communication channel between users.)

Finally, onion services are beneficial to the performance of the Tor network when compared to normal Tor circuits. This is due to the fact that they do not require exit relays, and therefore do not add to the saturation of their limited bandwidth. Facebook transparently redirects connections from Tor to one of its onion services—albeit reconfigured to only use a single hop on their half of the circuit—for this reason.

2.2 Threat Model

It is worth noting that the protections granted by Tor, whether on onion services or the typical usage for accessing the wider internet, are not unconditional. Notably, Tor does not defend against a global passive adversary—i.e., an adversary that can passively monitor any internet traffic it wishes. Such an adversary would be able to monitor all connections entering and leaving Tor relays, and using timing and packet sizes, trace connections from their source to their destination.

Such an adversary has never been demonstrated, and is unlikely to exist in practice. Even the NSA, at the time of the Snowden leaks, relied on exploiting application vulnerabilities in surveillance targets to deanonymize them rather than pure traffic analysis [Sch13]. However, the global passive adversary is less of a real threat so much as a threat model, useful for some systems, as it encompasses a large set of less powerful but more likely attacks. Notably, even without a global view of all traffic, any observer who can simultaneously view the network activity on the client and server side of a connection will still quickly be able to correlate the traffic timing and size to confirm that said client and server are communicating [Dan04]. Accordingly, any adversary

⁵<https://onionshare.org/>

who controls a fraction of the bandwidth weight of the Tor network will be able to perform such a correlation with a probability correlated with that fraction.

In practice, malicious relays that perform traffic correlation attacks are exceedingly rare. While malicious groups of Tor relays are sometimes discovered, they so far have not been positioned for general traffic correlation attacks—though one attack did make use of malicious relays for traffic correlation to identify onion service lookups [Din14]. Traffic confirmation attacks from outside the Tor network, on the other hand, have been used by law enforcement to identify suspects or gather evidence when they have already identified the source and destination of the traffic [Dal13; Fed12].

It is currently hoped that general traffic correlation attacks, in a form that would allow identifying the source and destination of a connection rather than confirmation, are prohibitively expensive for most threat models. In any case, any attempt to address such an adversary would necessarily require an increase in some combination of latency or bandwidth overhead [DMMK18].

2.3 Terminology

Finally, the name “Tor” comes with certain ambiguity—it is the name of the network, the project, the organization, the application, etc. Which of these is being referred to is hopefully clear from respective context throughout this thesis, but one minor hint is that when referring to the Tor application in particular—i.e., the executable run on a machine to run a Tor client, relay, or onion service—it is spelled in all lower-case (“tor”, sometimes referred to in spoken discussions as “little-t tor”).

Chapter 3

Conducting Statistically Sound Tor Experiments

In this chapter, we demonstrate tools and techniques for allowing researchers to perform statistically sound and scalable Tor network experiments. In particular, we focus on experimentation regarding the usability of Tor, in terms of network performance. The usability of the Tor network is fundamental to the security it can provide [DM06]; prior work has shown that real-world adversaries intentionally degrade usability to cause users to switch to less secure communication protocols [AAH13]. Good usability enables Tor to retain more users [FGK+10], and more users generally corresponds to better anonymity [ADS03]. Tor has made improvements in three primary usability components: (i) the design of the interface used to access and use the network (i.e., Tor Browser) has been improved through usability studies [COA07; LFM+17; NCC12]; (ii) the performance perceived by Tor users has improved through the deployment of new traffic scheduling algorithms [JTG+18; TG10]; and (iii) the network resources available for forwarding traffic has grown from about 100 Gbit/s to about 700 Gbit/s over the last 10 years [Tor24].

Researchers have contributed numerous proposals for improving Tor performance in order to support continued growth in the network, including those that attempt to improve Tor’s path selection [ABEG13; BW16; DLKS15; HSWM19; IAW19; JJJ+17; LSL16; MAEP20; RP17; RWJ+20; SDW15; WTBS13; YL15a], load balancing [GSH16; IBW18; JHK10; JJH+17; JJS13; MWS11], traffic admission control [AG13; DRPW20; GH12; GJH14; JGW+14; JSH12; JTG+18; KCU+19; LLW+17; YL15b], and congestion control mechanisms [ABG+11; GJH13]. At the time of this research, the standard practice when proposing a new mechanism for Tor was to run a *single experiment* with each recommended configuration of the mechanism and a *single experiment* with standard Tor. Measurements of a performance metric (e.g., download time) are taken during each experiment, the empirical distributions over which are directly compared across experiments.

Unfortunately, the experiments (typically simulations or emulations [SGD15]) were done in scaled-down Tor test networks that were sampled from the state of the true network at a static point in time [JBHD12]; only a *single sample* is considered even though in reality the network changes over time in ways that could change the conclusions. Moreover, statistical inference techniques (e.g., repeated trials and interval estimates) were generally not applied during the analysis of results, leading to questionable conclusions. Perhaps due in part to undependable results, only a few Tor performance research proposals have been deployed over the years [JTG+18; TG10] despite the abundance of available research.

Contributions In this work, we advanced what was the state of the art at the time (2020) by building foundations for conducting sound Tor performance research in three major ways: (i) we established an *ontology* of Tor experimentation, enumerating the elements of a Tor network a researcher may wish to control for or measure; (ii) we designed and validated Tor experimentation *models* and developed new and improved modeling and experimentation *tools* that together allow us to create and run more representative Tor test networks faster than was previously possible; and (iii) we developed *statistical methodologies* that enable sound statistical inference of experimentation results and demonstrate how to apply our methodologies through a case study on a concrete set of Tor experiments. In doing so, we expanded the ability of researchers to safely perform valid and useful Tor experiments, having since become the *de facto* standard for generating and configuring such experiments [Aro24; DDB23; DSC+22; DT23; JW23; Mic21; OAD22; PAKE23; RE22; WHH22].

Ontology We built foundations for Tor performance research by first establishing in Section 3.2 an extensible ontology of characteristics of a Tor network that a researcher could conceivably wish to measure or configure in an experiment. We identify various metrics that have been of use in previous research, and structure them in such a way to make their relationship to each other clear. Although not intended to be complete, our ontology forms a starting basis for understanding the elements a researcher may wish to query (on a test network or the actual network) and a means of understanding where other variables we did not enumerate would fit.

Models and Tools In Section 3.3 we present what was, at the time, a new Tor network modeling methodology that produces *more representative* Tor networks by considering the state of the network *over time* rather than at a static point as was previously standard [JBHD12]. We designed our modeling methodology to support the flexible generation of Tor network models with configurable network, user, traffic load, and process scale factors, supporting experiments in computing facilities with a range of available resources. We designed our modeling tools such

that expensive data processing tasks need only occur once, and the result has been distributed to the Tor community and used to efficiently generate many network models [Aro24; DDB23; DSC+22; DT23; JW23; Mic21; OAD22; PAKE23; RE22; WHH22].

In Section 3.4 we contribute new and improved experimentation tools that we optimized to enable us to run Tor experiments *faster* and at a *larger scale* than was previously possible. In particular, we describe several improvements we made to Shadow [JH12], the most popular and validated platform for Tor experimentation, and demonstrate how our Tor network models and improvements to Shadow increased the scalability of Tor simulations. We showcased these contributions by running the largest known Tor simulations—full-scale Tor networks with 6,489 relays and 792k simultaneously active users. We also ran smaller-scale networks of 2,000 relays and 244k users to compare to prior work: we observed a reduction in RAM usage of 1.7 TiB (64%) and a reduction in run time of 33 days, 12 hours (94%) compared to what was, at the time, the state of the art [JTH18].

Statistical Methodologies In Section 3.5 we describe a methodology that enables us to conduct sound statistical inference using the results collected from scaled-down (sampled) Tor networks. We found that running *multiple* simulations in *independently sampled* networks is necessary in order to obtain statistically significant results, a methodology that had never before been implemented in Tor performance research, rendering the conclusions drawn in previous work questionable (see Section 3.1.3). We describe how to use multiple networks to estimate the distribution of a random variable and compute confidence intervals over that distribution, and discuss how network sampling choices would affect the estimation.

In Section 3.6 we present a case study in order to demonstrate how to apply our modeling and statistical methodologies to conduct sound Tor performance research. We present the results from a total of 420 Tor simulations across three network scale and two traffic load factors. We find that the precision of the conclusions that can be drawn from the networks used for simulations are dependent upon the scale of those networks. Although it is possible to derive similar conclusions from networks of different scales, fewer simulations are generally required in larger-scale than smaller-scale networks to achieve a similar precision. We conclude that one simulation is never enough to achieve statistically significant results.

Availability Through this work we developed new modeling tools and improvements to Shadow that we released as open-source software as part of OnionTrace v1.0.0, TorNetTools v1.1.0, TGen v1.0.0, and Shadow v1.13.2.¹ We have made these and other research artifacts publicly

¹<https://github.com/shadow/{oniontrace,tornettools,tgen,shadow}>

available.² While the architecture of Shadow has gone on to change significantly since this research, the other tools remain largely the same after our changes made as part of this work.

3.1 Background and Related Work

We provide a brief background on Tor experimentation before describing prior work on Tor experimentation, modeling, and performance.

3.1.1 Tor Experimentation Tools

Early Tor experimentation tools included packet-level simulators that were designed to better understand the effects of Tor incentive schemes [JHK10; NDW10]. Although these simulators reproduced some of Tor’s logic, they did not actually use Tor source code and quickly became outdated and unmaintained. Recognizing the need for a more realistic Tor experimentation tool, researchers began developing tools following two main approaches: network emulation and network simulation [SGD15].

Network Emulation ExperimenTor [BSG11] is a Tor experimentation testbed built on top of the ModelNet [VYW+03] network emulation platform. ExperimenTor consists of two components that generally run on independent servers (or clusters): one component runs client processes and the other runs the ModelNet core emulator that connects the processes in a virtual network topology. The performance of this architecture was improved in SNEAC [Sin14] through the use of Linux Containers and the kernel’s network emulation module `netem`, while tooling and network orchestration were improved in NetMirage [Ung17]. With the exception of the new TIGER system—a Docker-based emulator for Tor networks that has so far only been used in its original publication [LCBS23]—these tools seem to have fallen out of favor for Tor experiments.

Network Simulation Shadow [JH12] is a hybrid discrete-event network simulator that runs applications as plugins. We provide more background on Shadow in Section 3.4.1. Shadow’s original design was improved with the addition of a user-space non-preemptive thread scheduler [MJ15], and later with a high performance dynamic loader [TJG18]. Additional contributions have been made through several research projects [JGW+14; JTG+18; JTH18], and we made further contributions that improve Shadow’s efficiency and correctness as described in Section 3.4.2. Since this work, other research has gone on to make further extensive changes [JNW22].

²<https://neverenough-sec2021.github.io>

3.1.2 Tor Modeling

An early approach to model the Tor network was developed for both Shadow and Experiment-Tor [JBHD12]. The modeling approach produced scaled-down Tor test networks by sampling relays and their attributes from a *single* true Tor network consensus. As a result, the models are particularly sensitive to short-term temporal changes in the composition of the true network (e.g., those that result from natural relay churn, network attacks, or misconfigurations). The new techniques we present in Section 3.3.2 are more robust to such variation because they are designed to use Tor metrics data spanning a user-selectable time period (i.e., from any chosen set of consensus files) in order to create simulated Tor networks that are more *representative* of the true Tor network over time.

In previous models, the number of clients to use and their behavior profiles were unknown, so finding a suitable combination of traffic generation parameters that would yield an appropriate amount of background traffic was often a challenging and iterative process. But with the introduction of privacy-preserving measurement tools [EDG14; FMJS17; JJ16; MS17] and the available Tor measurement studies [JJ16; JTH18; MWJ+18], we have gained a more informed understanding of the traffic characteristics of Tor. Our new modeling techniques use Markov models informed by (privacy-preserving) statistics from true Tor traffic [JTH18], while significantly improving experiment scalability as we demonstrate in Section 3.4.3.

3.1.3 Tor Performance Studies

The Tor experimentation tools and models described above have assisted researchers in exploring the various bodies of Tor research mentioned at the beginning of the chapter, as well as denial of service mechanisms [CS14; Hop14; JTJS14; JVS19; RP18], and how these proposed changes or attacks affect Tor performance and security [AG16]. It is this prior work that overwhelmingly relied on single scaled-down Tor network models. Although some studies used multiple trials of each experimental configuration in the *chosen* sampled network [JJH+17; JJS13], none of them involved running experiments in *multiple* sampled networks, which is necessary to estimate effects on the real-world Tor network (see Section 3.5). Additionally, statistical inference techniques (e.g., interval estimates) are not applied during the analysis of the results, leading to questions about the extent to which the conclusions drawn in previous work are relevant to the real world. Our work advanced the state of the art of the experimental process for Tor performance research: in Section 3.5 we describe our statistical methodologies that enable researchers to conduct sound statistical inference from Tor experimentation results, and in Section 3.6 we present a case study to demonstrate how to put our methods into practice.

3.2 Ontology

In this section, we describe an ontology of the Tor network, from the perspective and for the purpose of controlled performance research. While our ontology shows one way of orienting known factors to consider when conducting Tor experiments, we emphasize that it is not intended to be complete, and is merely the product of this author’s knowledge of past Tor research and research platforms. The most interesting future research may come not from the measurement of properties (or even elements) listed here, but from the gaps and undervalued areas that are currently unexplored in Tor research.

Ontology The ontology consists of elements (clients, relays, servers, and the network), each of which have properties that can be further recursively subdivided into (sub)properties. These properties can be viewed as the variables of an experiment, and therefore can be separated into independent and dependent variables. Independent variables are properties that are *set*, chosen during the course of experiment configuration (e.g., the number of clients, or available bandwidth). Dependent variables are properties that can be *measured* as results of the experiment (e.g., throughput, or successful downloads). The division between what constitutes independent and dependent variables depends on the specific context of an experiment. In this ontology, we classified properties based on experimentation platforms for controlled Tor experiments. More observational research (e.g., the measurement studies mentioned in Section 3.1.2, or the measurements done on Tor Metrics [Tor24]) would have a different perspective, primarily manifesting as many properties shifting from independent to dependent variables. Even with this particular point of reference, however, some properties can concurrently exist as both independent and dependent variables in one experiment. Packet loss, for example, is something that can be configured as a property of the network (i.e., a particular link can be configured to drop packets with some probability), but will also occur as a result of the natural behavior of TCP stacks establishing a stable connection and can therefore be measured.

The rest of this section is dedicated to describing the elements of our ontology. The properties of these elements are enumerated and classified in Table 3.1. While most of the terms are self-explanatory, they are also briefly described in Table 3.2.

Network The network element represents the connections between other elements, as well as meta-properties that are not directly measurable on individual nodes (though analogues may be). Latency and bandwidth, for example, are properties directly instantiated in the links between the other elements. The time to a steady state, on the other hand, is something that can be measured,

Table 3.1: Classification of the experimentation properties in our Tor ontology into Independent and Dependent variables, organized by element. An arrow indicates a subproperty.

	latency	jitter	bandwidth	reliability	packet loss	congestion	path/routing	time to consensus	IP address	geolocation	OS/kernel stack	hardware CPU/memory	throughput	control usage	retransmissions	behavior model	number of connections	duration of connections	idle time	time to first/last byte	relay selection algorithm	max num. open circuits	circuit build time	errors	Tor relay configuration	packets delay	congestion	processing (Tor)	connections (sock)	sockets open	streams open	connecting clients	errors	behavior model	port																	
Indep.	●	●	●	●	-	-	-	-	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●																
Dep.	-	-	-	●	●	●	●	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-																
	Network							Common							Clients								Relays						Servers																							
																															Network Nodes																					

but not as an actual property of any particular element, so much as a measurement of a constructed representation of the entire network.

Network Nodes Network nodes are all endpoints in the network (i.e., every client, relay, and server). While we could assign their common properties to each of the elements discussed in the remainder of this section, we group them together to reflect their commonality (and to avoid redundancy).

Some properties, such as control overhead, could arguably be positioned as part of the network itself, but are in this ontology considered part of the network nodes. The deciding factor was whether the variable could be directly configured or measured as a property of a particular node. For example, while packet loss requires knowledge of the link between two relays, control overhead can be measured on only one end; therefore, we place the former as a network property and the latter as a property of the network node. From a more empirical perspective, tools such as Shadow and emulators would configure/measure packet loss on the edges of a network graph, while control overhead would be measured using data collected from the node.

Clients Clients are the subclass of network nodes that run applications proxied via Tor; they represent both normal Tor clients, as well as onion services. Client properties include those relating to the Tor application itself, as well as the application(s) being proxied through it.

Relays Relays are the subclass of network nodes that run Tor relays. As above, relay properties include those of the Tor application, as well as the environment in which it runs.

Table 3.2: Description of properties from the ontology. Arrows denote subproperties.

		Property	Description
Network	Network	latency	The amount of time for a packet to traverse from one network node to another.
		↳ jitter	The variation in latency.
		bandwidth	The amount of data a network connection can transfer in a given amount of time.
		reliability	The probability of a network connection successfully transferring data.
		↳ packet loss	The probability of a packet on an existing connection not arriving at the destination.
		path/routing	The set of network nodes a packet passes through to arrive at its destination.
		congestion	The amount of traffic load exceeding the capacity of the link or network node.
		time to network consensus	The amount of time until the Tor network generates a valid network consensus file.
		time to steady state	The amount of time until the network displays consistent behavior.
Network Nodes	Common	quantity	The amount of this particular type of node in the network.
		IP address	The external IP address of the node.
		geolocation	Where the node is geographically located.
		stack	What Tor and associated processes are running on.
		↳ OS/kernel	The operating system, especially the network stack.
		↳ hardware	The computer components and their characteristics, such as CPU speed or memory capacity.
		CPU/memory usage	The amount of CPU time and RAM used.
		throughput	The total network traffic over time, including overhead (e.g., headers and retransmissions).
		goodput	The total usable traffic over time, thus not including overhead (e.g., headers or retransmissions).
		control overhead	The amount of traffic that is spent on protocol data, rather than payload data.
		↳ retransmissions	The amount of traffic that was duplicated as a result of TCP acknowledgements not being received (in time).
	Clients	behavior model	How the client behaves.
		↳ number of connections	How many network connections the client creates (typically to servers, via Tor relays).
		↳ duration of connections	How long network connections last before being closed.
		↳ traffic type	The protocol and traffic properties (e.g., web pages, large downloads).
		↳ idle time	The time spent not sending any traffic, either because there is nothing being sent over a currently active connection, or because the client has no active connections.
		time to first byte	The amount of time to receive the first byte of a download. A.K.A. round trip time (RTT).
		time to last byte	The amount of time to complete a download.
		Tor	
		↳ relay selection algorithm	How Tor chooses which relays to route through.
		↳ max number of open circuits	The maximum number of circuits simultaneously open.
		↳ max duration of circuits	The maximum amount of time circuits remain open.
		↳ circuit build time	How long it takes to construct a circuit.
		↳ errors	The number and characteristics of errors encountered.
	Relays	Tor relay configuration	The configuration of the relay via configuration files or changes to the Tor application.
		packet delay	The amount of additional time for a packet to enter and leave the entire relay.
		congestion	Network congestion specifically as a result of the Tor relay process.
		processing (Tor)	The amount of time spent processing packets within the Tor process.
		processing (stack)	The amount of time spent processing packets outside the Tor process (primarily the OS).
		connections (number of:)	
		↳ sockets open	The number of network sockets the relay has open.
		↳ streams and circuits	The number of TCP streams and Tor circuits.
	Servers	↳ connecting clients	The number of clients that connect to this relay.
		↳ errors	The number and characteristics of errors encountered.
	Servers	behavior model	How the server interacts with the client application communicating with it.
		port	The network ports the server is listening on. This is distinct from the behavior model in that Tor relays interact with it (via exit policies), not just the client.

Servers Servers are the subclass of network nodes that represent non-Tor network entities; e.g., web servers and non-Tor clients. Because they do not run Tor, and will typically be creating requests or responding to requests created elsewhere, they add few properties not already captured above.

3.3 Models for Tor Experimentation

In order to conduct Tor experiments that produce meaningful results, we must have network and traffic models that accurately represent some set of the elements and properties of the above ontology. In this section, we describe the modeling techniques we introduced to make use of the data from privacy-preserving measurement studies [JJ16; JTH18; MWJ+18]. Note that while exploring alternatives for every modeling choice that will be described in this section is out of scope for this thesis, we will discuss some alternatives that are worth considering in Section 3.7.

3.3.1 Internet Model

Network communication is vital to distributed systems; the bandwidth and the network latency between nodes are primary characteristics that affect performance. Jansen *et al.* have produced an Internet map [JTH18] that we find useful for our purposes; we briefly describe how it was constructed before explaining how we modify it.

To produce an Internet map, Jansen *et al.* [JTH18] conducted Internet measurements using globally distributed vantage points (called *probes*) from the RIPE Atlas measurement system (atlas.ripe.net). They assigned a representative probe for each of the 1,813 cities in which at least one probe was available. They used `ping` to estimate the latency between all of the 1,642,578 distinct pairs of representative probes, and they crawled speedtest.net to extract upstream and downstream bandwidths for each city.³ They encoded the results into an Internet map stored in the graphml file format; each vertex corresponds to a representative probe and encodes the bandwidth available in that city, and each edge corresponds to a path between a pair of representative probes and encodes the network latency between the pair.

Also encoded on edges in the Internet map were packet loss rates. Each edge e was assigned a packet loss rate p_e according to the formula $p_e \leftarrow 0.015 \cdot L_e / 300$ where L_e is the latency of edge e . This improvised formula was not based on any real data. Our experimentation platform (described in Section 3.4) already includes for each host an edge router component that drops packets when

³speedtest.net ranks mobile and fixed broadband speeds around the world.

Table 3.3: Statistics computed during the *staging* phase.

Stat.	Description
r_i	the fraction of network consensuses in which relay i was running
g_i	the fraction of network consensuses in which relay i was a guard relay
e_i	the fraction of network consensuses in which relay i was an exit relay
w_i	the median normalized network consensus weight of relay i
b_i	the max observed bandwidth of relay i
λ_i	the median bandwidth rate of relay i
β_i	the median bandwidth burst of relay i
C_ρ	median across network consensuses of relay count for each position $\rho^\dagger$
W_ρ	median across network consensuses of total weight for position $\rho^\dagger$
U_c	median normalized probability that a user is in country $c^\ddagger$

[†] Valid positions are D : exit+guard, E : exit, G : guard, and M : middle.

[‡] Valid countries are any two-letter country code (e.g., *us*, *ca*, etc.).

buffers are full. Because additional packet loss from core routers is uncommon [LSB08],⁴ we modify the Internet map by setting p_e to zero for all edges.⁵ We use the resulting Internet model in all Tor simulations in this thesis.

3.3.2 Tor Network Model

To the Internet model we add hosts that run Tor relays and form a Tor overlay network. The Tor modeling task is to choose host bandwidths, Internet locations, and relay configurations that support the creation of Tor test networks that are representative of the true Tor network.

We construct Tor network models in two phases: *staging* and *generation*. The two-phase process allows us to perform the computationally expensive staging phase once, and then perform the computationally inexpensive generation phase any number of times. It also allowed us to release the staging files to the community, whose members can use our Tor modeling tools without needing to first process large datasets.

⁴A notable exception is China, which sees high packet loss only on packets *entering* the country [ZMW+20]. As China attempts to censor Tor, no relays are hosted there, but some clients do use Tor’s anti-censorship functionality.

⁵Future work should consider developing a more realistic packet loss model that is, e.g., based on measurements of actual Tor clients and relays.

Staging

Ground truth details about the temporal composition and state of the Tor network are available in Tor network data files (i.e., hourly network consensus and daily relay server descriptor files), which have been published since 2007. We first gather the subset of these files that represents the time period that we want to model (e.g., all files published in January 2019), and then extract network attributes from the files in the staging phase so that we can make use of them in the networks we later generate. In addition to extracting the IP address, country code, and fingerprint of each relay i , we compute the per-relay and network summary statistics shown in Table 3.3. We also process the Tor *users* dataset containing per-country user counts, which Tor has published daily since 2011 [Tor24]. From this data we compute the median normalized probability that a user appears in each country. We store the results of the staging phase in two small JSON files (a few MiB each) that we use in the generation phase. Note that we could make use of other network information if it were able to be safely measured and published (see Section 3.2 for examples of independent variables that could be useful).

Generation

In the generation phase, we use the data extracted during the staging phase and the results from a privacy-preserving Tor measurement study [JTH18] to generate Tor network models of a configurable *scale*. For example, a *100% Tor network* represents a model of equal scale to the true Tor network. Each generated model is stored in a *configuration file* (at the time as XML, now YAML), which specifies the hosts that should be instantiated, their bandwidth properties and locations in the Internet map, the processes they run, and configuration options for each process. Instantiating a model will result in a Tor test network that is representative of the true Tor network. We describe the generation of the configuration file by the type of hosts that make up the model: *Tor network relays*, *traffic generation*, and *performance benchmarking*.

Tor Network Relays The relay staging file may contain more relays than we need for a 100% Tor network (due to relay churn in the network during the staged time period), so we first choose enough relays for a 100% Tor network by sampling $n \leftarrow \sum_{\rho} C_{\rho}$ relays without replacement, using each relay’s running frequency r_i as its sampling weight.⁶ We then assign the guard and exit flag to each of the n sampled relays j with a probability equal to the fraction of network consensuses in which relay j served as a guard g_j and exit e_j , respectively.

⁶Alternatives to weighted sampling should be considered if staging time periods during which the Tor network composition is extremely variable.

To create a network whose scale is $0 < s \leq 1$ times the size of the 100% network,⁷ we further subsample from the sampled set of n relays to use in our scaled-down network model. We describe our subsampling procedure for middle relays for ease of exposition, but the same procedure is repeated for the remaining positions (see Table 3.3 note[†]). To subsample $m \leftarrow s \cdot C_M$ middle relays,⁸ we: (i) sort the list of sampled middle relays by their normalized consensus weight w_j , (ii) split the list into m buckets, each of which contains as close as possible to an equal number of relays, and (iii) from each bucket, select the relay with the median weight w_j among those in the bucket. This strategy guarantees that the weight distribution across relays in our subsample is a best fit to the weight distribution of relays in the original sample [JBHD12].

A new host is added to the configuration file for each subsampled relay k . Each host is assigned the IP address and country code recorded in the staging file for relay k , which will allow it to be placed in the nearest city in the Internet map. The host running relay k is also assigned a symmetric bandwidth capacity equal to b_k ; i.e., we use the maximum observed bandwidth as our best estimate of a relay’s bandwidth capacity. Each host is configured to run a Tor relay process that will receive the exit and guard flags that we assigned (as previously discussed), and each relay k sets its token bucket rate and burst options to λ_k and β_k , respectively. When executed, the relay processes will form a functional Tor overlay network capable of forwarding user traffic.

Traffic Generation A primary function of the Tor network is to forward traffic on behalf of users. To accurately characterize Tor network usage, we use the following measurements from the aforementioned privacy-preserving Tor measurement study [JTH18]: the total number of active users $\phi = 792\text{k}$ (counted at guard relays) and the total number of active circuits $\psi = 1.49\text{M}$ (counted at exit relays) in an average 10 minute period.

To generate a Tor network whose scale is $0 < s \leq 1$ times the size of the 100% network, we compute the total number of users we need to model as $u \leftarrow s \cdot \phi$. We compute the total number of circuits that those u users create every 10 minutes as $c \leftarrow \ell \cdot s \cdot \psi$, where $\ell \geq 0$ is a load factor that allows for configuration of the amount of traffic load generated by the u users ($\ell = 1$ results in “normal” traffic load). We use a process scale factor $0 < p \leq 1$ to allow for configuration of the number of Tor client processes that will be used to generate traffic on the c circuits from the u users. Each of $p \cdot u$ Tor client processes will support the combined traffic of $1/p$ users, i.e., the traffic from $\tau \leftarrow c/(p \cdot u)$ circuits.

⁷Because of the RAM and CPU requirements (see Section 3.4), we expected that it would typically be infeasible to run 100% Tor networks. The configurable scale s allows for tuning the amount of resources required to run a model. For technical reasons described in Section 4.4.4, 100% Tor networks have since become likely impossible in Shadow.

⁸Technically, the C_M used at this stage is the number of relays in that position from the 100% network above, which will be very close to the real C_M in expectation, but not identical.

The p factor can be used to significantly reduce the amount of RAM and CPU resources required to run our Tor model; e.g., setting $p = 0.5$ means we only need to run half as many Tor client processes as the number of users we are simulating.⁹ At the same time, p is a reciprocal factor with respect to the traffic that each Tor client generates; e.g., setting $p = 0.5$ causes each client to produce twice as many circuits (and the associated traffic) as a single user would.

We add $p \cdot u$ new traffic generation client hosts to our configuration file. For each such client, we choose a country according to the probability distribution U , and assign the client to a random city in that country using the Internet map in Section 3.3.1.¹⁰ Each client runs a Tor process in client mode configured to disable guard relays¹¹ and a TGen traffic generation process that is configured to send its traffic through the Tor client process over localhost (we significantly extend a previous version of TGen [JTH18, §5.1] to support our models). Each TGen process is configured to generate traffic using Markov models (as we describe below), and we assign each host a bandwidth capacity equal to the maximum of $10/p$ Mbit/s and 1 Gbit/s to prevent it from biasing the traffic rates dictated by the Markov models when generating the combined traffic of $1/p$ users. Server-side counterparts to the TGen processes are also added to the configuration file (on independent hosts).

Each TGen process uses three Markov models to accurately model Tor traffic characteristics: (i) a *circuit* model, which captures the circuit inter-arrival process on a per-user basis; (ii) a *stream* model, which captures the stream inter-arrival process on a per-circuit basis; and (iii) a *packet* model, which captures the packet inter-arrival process on a per-stream basis. Each of these models are again based on the privacy-preserving measurement study that used PrivCount [JJ16] to collect measurements of real traffic being forwarded by a set of Tor exit relays [JTH18]. We encode the *circuit* inter-arrival process as a simple single state Markov model that emits new circuit events according to an exponential distribution with rate τ/μ microseconds, where $\mu \leftarrow 6 \cdot 10^8$ is the number of microseconds in 10 minutes. New streams on each circuit and packets on each stream are generated using the *stream* and *packet* Markov models, respectively, which were directly measured in Tor and published in previous work [JTH18, §5.2.3].

The rates and patterns of the traffic generated using the Markov models will mimic the rates and patterns of real Tor users: the models encode common distributions (e.g., exponential and log-normal) and their parameters, such that they can be queried to determine the amount of time

⁹A primary effect of $p < 1$ is fewer network consensus fetches, the network impact of which is negligible relative to the total traffic generated.

¹⁰Shadow will arbitrarily choose an IP address for the host such that it can route packets to all other simulation hosts (clients, relays, and servers).

¹¹Although a Tor client uses guard relays by default, for us it would lead to inaccurate load balancing because each client simulates $1/p$ users. Support in the Tor client for running multiple $(1/p)$ parallel guard relay “sessions” (i.e., assigning a guard relay to each user “session”) is an opportunity for future work.

to wait between the creation of new circuits and streams and the transfer of packets (in both the send and receive directions).

Each TGen client uses unique seeds for all Markov models so that it generates unique traffic characteristics.¹² Each TGen client also creates a unique SOCKS username and password for each generated circuit and uses it for all Tor streams generated in the circuit; due to Tor’s `IsolateSOCKSAuth` feature, this ensures that streams intended for different circuits will in fact be assigned to independent circuits.

We highlight that although prior work also made use of the *stream* and *packet* Markov models [JTH18, §5.2.3], we extend previous work with a *circuit* Markov model that can be used to continuously generate circuits independent of the length of an experiment. Moreover, previous work did not consider either load scale ℓ or process scale p ; ℓ allows for research under varying levels of congestion, and our optimization of simulating $1/p$ users in each Tor client process allows us to more quickly run significantly larger network models than we otherwise could (as we will show in Section 3.4.3).

Performance Benchmarking The Tor Project has published performance benchmarks since 2009 [Tor24]. The benchmark process downloads 50 KiB, 1 MiB, and 5 MiB files through the Tor network several times per hour, and records various statistics about each download including the time to download the first and last byte of the files. We mirror this process in our models; running several benchmarking clients that use some of the same code as Tor’s benchmarking clients (i.e., TGen) allows us to *directly* compare the performance obtained in our simulated Tor networks with that of the true Tor network.

Modeling Tools

We implemented several tools that are fundamental to readily model and execute realistic Tor test networks. We released these tools as open source software to help facilitate Tor research: (i) a Tor network modeling toolkit called TorNetTools (3,034 LoC) that implements our modeling algorithms from Section 3.3.2; (ii) extensions and enhancements to the TGen traffic generator [JTH18, §5.1] (6,531 LoC added/modified and 1,411 removed) to support our traffic generation models; and (iii) a tool called OnionTrace (2,594 LoC) to interact with a Tor process and improve reproducibility of experiments.

¹²The Markov model seeds are unique across clients, but generated from the same master seed in order to maintain a deterministic simulation.

3.4 Tor Experimentation Platform

The models that we described in Section 3.3 could reasonably be instantiated in a diverse set of experimentation platforms in order to produce representative Tor test networks. We use Shadow [JH12], the most popular and validated platform for Tor experimentation. We provide a brief background on Shadow’s design, explain the improvements we made to support accurate experimentation, and show how our improvements and models from Section 3.3 contribute to the state of the art.

3.4.1 Shadow Background

Shadow is a hybrid experimentation platform [JH12]. At its core, Shadow is a conservative-time discrete-event network simulator: it simulates hosts, processes, threads, TCP and UDP, routing, and other kernel operations. One of Shadow’s advantages is that it runs real applications and directly executes them as native code (at the time of this research, as dynamically loaded plugins; as of the latest Shadow release, as independent but managed processes). In this regard, Shadow emulates a network and a Linux environment: applications running in a simulation should function as they would if they were executed on a bare-metal Linux installation.

Because Shadow is a user-space, single user application, it can easily run on laptops, desktops, and servers with minimal configuration (resource requirements depend on the size of the experiment model). As a simulator, Shadow has complete control over simulated time; experiments may run faster or slower than real time depending on: (i) the simulation load relative to the processing resources available on the host machine, and (ii) the inherent parallelizability of the experiment model. This control over time decouples the fidelity of the experiment from the processing time required to execute it, and allows Shadow to scale independently of the processing capabilities of the host machine; Shadow is usually limited by the RAM requirements of its loaded plugins.¹³ The combination of its features makes Shadow a powerful tool for Tor experimentation, and has made it the most popular and standard tool for conducting Tor performance research for some time [SGD15].

3.4.2 Shadow Improvements

After investigation of the results from some early experiments, we made several improvements to Shadow that enable faster experiments and produce significantly more accurate results when

¹³Sufficiently large experiments can run into other limitations—namely, any severe computational bottlenecks in the running applications, and, in the latest versions of Shadow, maximum process counts on Linux.

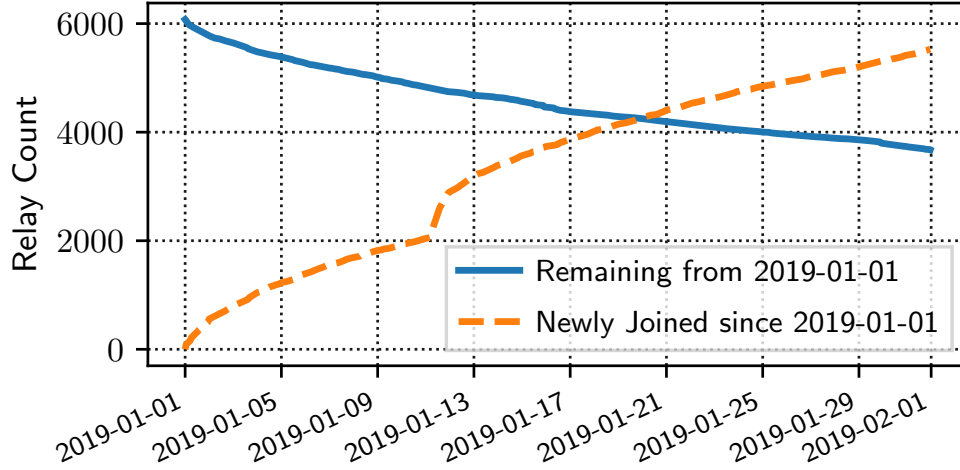


Figure 3.1: The rate of Tor relay churn over all 744 network consensuses from January 2019. Shown are the number of Tor relays that existed on 2019-01-01 that remain and the number of relays that did not exist on 2019-01-01 that joined (and possibly left again) over time.

running our Tor network models from Section 3.3.2. Our improvements include run-time optimizations, fixes to ensure deterministic execution, faster Tor network bootstrapping, more realistic TCP connection limits, and several network stack improvements. Our improvements were incorporated into Shadow v1.13.2.

3.4.3 Evaluation

This chapter has presented two types of foundational contributions thus far: those that result in more *representative Tor networks*, and those that allow us to run more *scalable simulations* faster than was previously possible. We demonstrate these contributions through Tor network simulations in Shadow.

Representative Networks We produce more representative networks by considering the state of the network over time rather than modeling a single snapshot as did previous work [JBHD12; JTH18]. We consider relay churn to demonstrate how the true Tor network changes over time. Figure 3.1 shows the rate of relay churn over all 744 network consensus files (1 per hour) in Tor during January 2019. After 2 weeks, fewer than 75% of relays that were part of the network on 2019-01-01 still remain while more than 3,000 new relays joined the network. After 3 weeks,

more new relays had joined the network than had remained since 2019-01-01. Our models account for churn by sampling from *all* such relays as described in Section 3.3.2.

In addition to producing more representative models, our Shadow network stack enhancements further improve network accuracy. To demonstrate these contributions, we simulate ten Tor network models that were generated following the methods in Section 3.3.2 (using Tor network state from 2019-01). We model Tor at the same $s = 0.31$ scale that was used in previous work [JTH18] (i.e., $\approx 2k$ relays and $\approx 250k$ users) using a process scale factor of $p = 0.01$ (i.e., each TGen process simulated $1/0.01 = 100$ Tor users). We compare our simulation results to those produced by state-of-the-art methods [JTH18] (which used Tor network state from 2018-01) and to reproducible Tor metrics [Tor23; Tor24] from the corresponding modeling years (2019 for our work, 2018 for the CCS 2018 work).¹⁴

The results in Figure 3.2 generally show that previous work is noticeably less accurate when compared to Tor 2018 than our work is compared to Tor 2019. We notice that previous work exhibited a high client download error rate in Figure 3.2c and significantly longer download times in Figures 3.2e–3.2g despite the network being appropriately loaded as shown in Figure 3.2h. We attribute these errors to the connection limit and network stack limitations that were present in the CCS 2018 version of Shadow (the errors are not present in this work due to our Shadow improvements from Section 3.4.2). The download error rate shows larger confidence intervals than other statistics (here as well as other figures) because errors are discrete rather than continuous, and happen in small absolute numbers. Also, we remark that the relay goodput in Figure 3.2h exhibits more variance in Tor than in Shadow because the Tor data is being aggregated over a longer time period (1 year for Tor vs. less than 1 hour for Shadow) during which the Tor network composition is significantly changing (see Figure 3.1).

Scalable Simulations Our new models and Shadow enhancements enable researchers to run larger networks faster than was previously possible. We demonstrate our improvements to scalability in two ways. First, we compare in the top part of Table 3.4 the resources required for the 31% experiments described above. We distinguish total run time from the time required to bootstrap all Tor relays and clients, initialize all traffic generators, and reach steady state. We reduced the time required to execute the bootstrapping process by 2 days, 18 hours, or 80%, while we reduced the total time required to run the bootstrapping process plus 25 simulated minutes of steady state by 33 days, 12 hours, or 94%. The ratio of real time units required to execute each simulated time unit during steady state (i.e., after bootstrapping has completed) was reduced by 96%, further highlighting our achieved speedup. When compared to models of the same

¹⁴Although the models used Tor data spanning one month, we consider it reasonable to reflect the general state of Tor throughout the respective year.

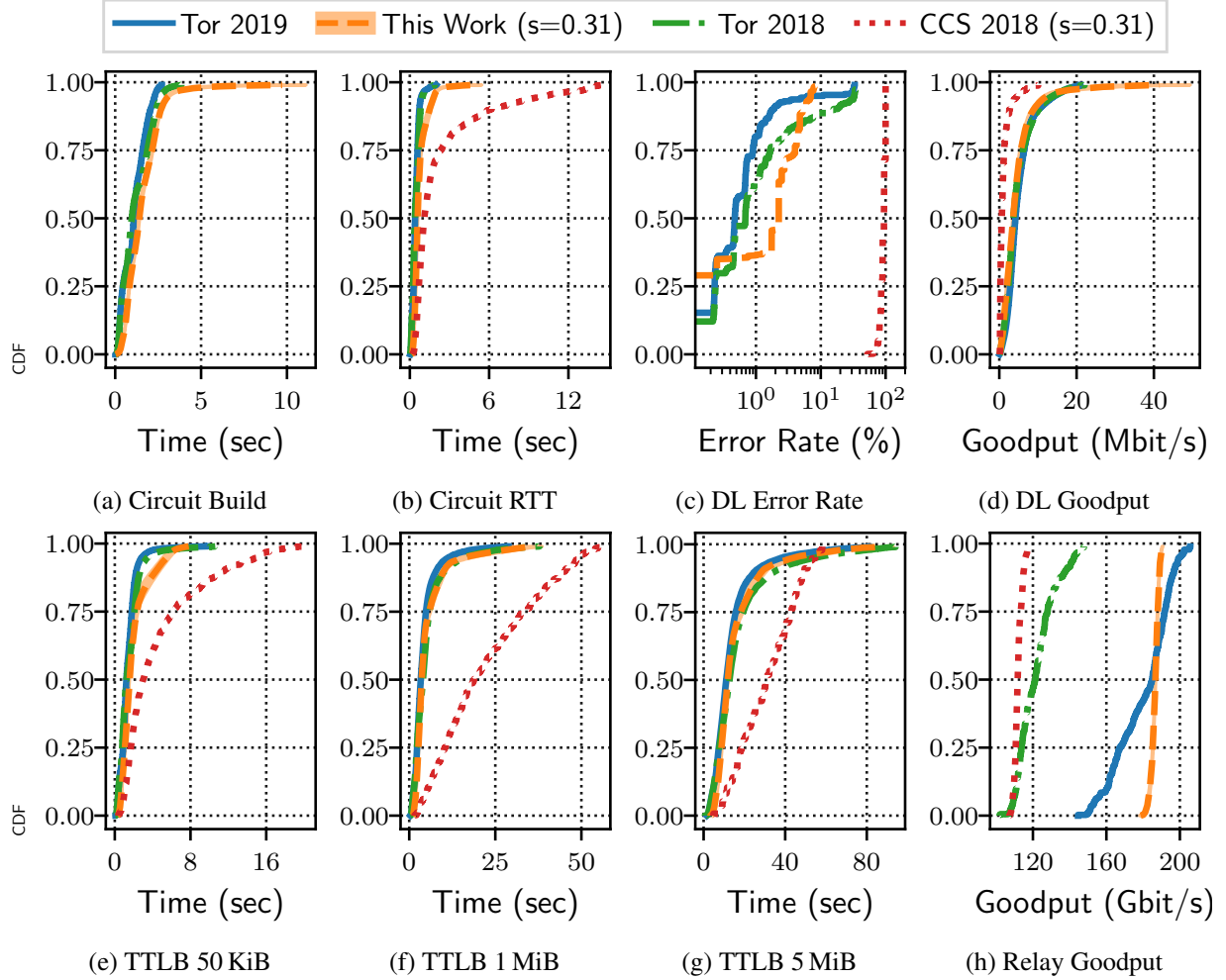


Figure 3.2: A comparison of simulated results with Tor metrics. Results are from 10 simulations at network scale $s = 0.31$ (modeled using Tor network state from 2019-01) and 1 simulation using methods from CCS 2018 [JTH18] (modeled using Tor network state from 2018-01) compared to reproducible Tor metrics [Tor23] during the respective years. Shown are benchmark client metrics for: (a) circuit build times; (b) round trip times (time from data request to first byte of response); (c) download error rate; (d) download goodput (i.e., transfer rate for range [0.5 MiB, 1 MiB] over 1 MiB and 5 MiB transfers), and (e)–(g) download times for transfers of size 50 KiB, 1 MiB, and 5 MiB. Relay goodput in (h) is, for each second, the sum over all relays of application bytes written (extrapolated by a $1/0.31$ factor to account for scale). (Note that circuit times in (a) are unavailable in the CCS 2018 model [JTH18].) The shaded areas represent 95% confidence intervals (CIs) that were computed following our method from Section 3.5.

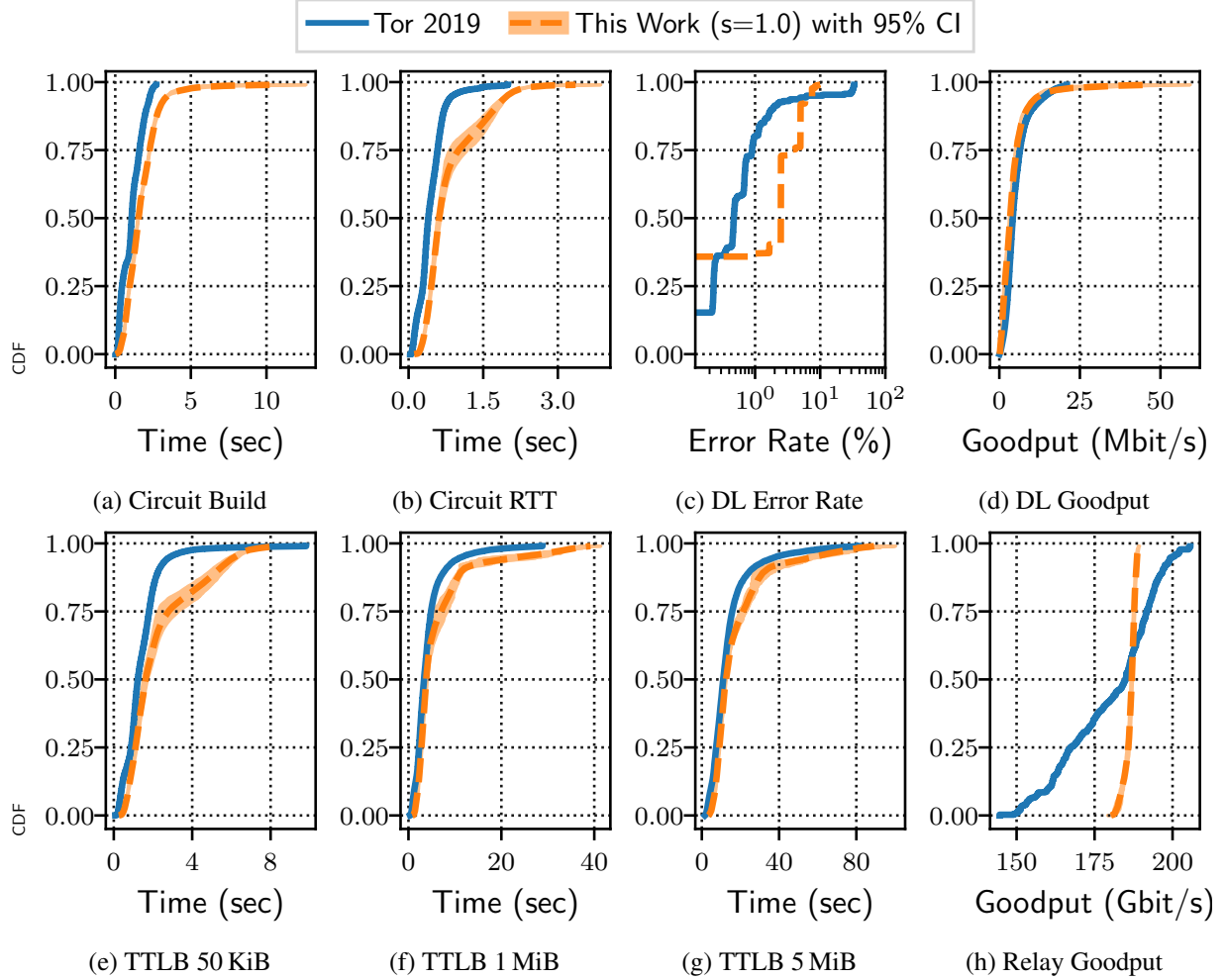


Figure 3.3: Results from 3 simulations at network scale $s = 1.0$. Modeled using Tor network state from 2019-01, compared to reproducible Tor metrics [Tor23]. The metrics are as were defined in the Figure 3.2 caption. The shaded areas represent 95% confidence intervals (CIs) that were computed following our method from Section 3.5.

Table 3.4: Scalability improvements over the prior state of the art

Model	Scale s^*	RAM ^{**}	Bootstrap Time ^{**}	Total Time ^{**}	Ω°	Networks
CCS'18 [JTH18] [†]	31%	2.6 TiB	3 days, 11 hrs.	35 days, 14 hrs.	1850	1
This work [†]	31%	932 GiB	17 hrs.	2 days, 2 hrs.	79	10
This work [‡]	100%	3.9 TiB	2 days, 21 hrs.	8 days, 6 hrs.	310	3

* **31%**: $\approx 2k$ relays and $\approx 250k$ users; **100%**: 6,489 relays and 792k users

** Maximum across networks.

$^\circ \Omega$: ratio of real time / simulated time in steady state (after bootstrapping)

[†] Using 8×10 -core Intel Xeon E7-8891v2 CPUs each running @3.2 GHz.

[‡] Using 8×18 -core Intel Xeon E7-8860v4 CPUs each running @2.2 GHz.

$s = 31\%$ scale from previous work, we observed that our improvements reduced the maximum RAM required to run bootstrapping plus 25 simulated minutes of steady state from 2.6 TiB down to 932 GiB (a total reduction of 1.7 TiB, or 64%).

Second, we demonstrate how our improvements enable us to run significantly larger models by running three Tor models at scale $s = 1.0$, i.e., at 100% of the size of the true Tor network. We are the first to simulate Tor test networks of this scale.¹⁵ The bottom part of Table 3.4 shows that each of our 100% Tor networks consumed at most 3.9 TiB of RAM, completed bootstrapping in 2 days, 21 hours, and ran the entire simulation (bootstrapping plus 25 simulated minutes of steady state) in 8 days, 6 hours. We show in Figure 3.3 that our 100% networks also achieve similar performance compared to the metrics published by Tor [Tor23]. Our results are plotted with 95% confidence intervals to better understand how well our sampling methods are capable of reproducing the performance characteristics of the true Tor network. We describe how to conduct such a statistical inference in Section 3.5 next.

3.5 On the Statistical Significance of Results

Recall that our modeling methodology from Section 3.3.2 produces *sampled* Tor networks at scales of $0 < s \leq 1$ times the size of a 100% network. Because these networks are sampled using data from the true Tor network, there is an associated sampling error that must be quantified when making predictions about how the effects observed in sampled Tor networks generalize to the true Tor network. In this section, we establish a methodology for employing statistical

¹⁵We attempted to run a 100% scale Tor network using the CCS 2018 model [JTH18], but it did not complete the bootstrapping phase within 30 days.

inference to quantify the sampling error and make useful predictions from sampled networks. In our methodology, we: (i) use repeated sampling to generate multiple sampled Tor networks; (ii) estimate the true distribution of a random variable under study through measurements collected from multiple sampled network simulations; and (iii) compute statistical confidence intervals to define the precision of the estimation. We show that multiple sampled networks are needed (as opposed to multiple trials in the same network) in order to gain information about the true Tor network, and we discuss how the number of samples per network and the number of sampled networks affect the inference process.

We remark that it is paramount to conduct a statistical inference when running experiments in sampled Tor networks in order to contextualize the results they generate. Our methodology employs confidence intervals (CIs) to establish the precision of estimations that are made across sampled networks. CIs allow researchers to make a statistical argument about the extent to which the results they have obtained are relevant to the real world. As we will demonstrate in Section 3.6, CIs help guide researchers to sample additional Tor networks (and run additional simulations) if necessary for drawing a particular conclusion in their research. Our methodology represents a shift in the state of the art of analysis methods typically used in Tor network performance research, which had previously ignored statistical inference and CIs altogether (see Section 3.1.3).

3.5.1 Methodology

When conducting research using experimental Tor networks, suppose we have an interest in a particular network metric; for example, our research might call for a focus on the distribution of time to last byte across all files of a given size downloaded through Tor as an indication of Tor performance (see our ontology in Section 3.2 for examples of other useful metrics). Because the values of such a variable are determined by the outcomes of statistical experiments, we refer to the variable as random variable X . The true probability distribution over X is $P(X)$, the true cumulative distribution is $F_X(x) = P(X \leq x)$, and the true inverse distribution at quantile y is $F_X^{-1}(y)$ such that $y = F_X(F_X^{-1}(y))$. Our goal is to estimate $P(X)$ (or equivalently, F_X and F_X^{-1}), which we do by running many simulations in sampled Tor networks and averaging the empirical distributions of X at a number of quantiles across these simulations. Table 3.5 summarizes the symbols that we use to describe our methodology.

Repeated Sampling A single *network* sampled from the true Tor network may not consistently produce perfectly representative results due to the sampling error introduced in the model sampling process (i.e., Section 3.3). Similarly, a single *simulation* may not perfectly represent a sampled network due to the sampling error introduced by the random choices made in the simulator (e.g.,

Table 3.5: Symbols used to describe our statistical methodology.

Symbol	Description
$P(X)$	true probability distribution of random variable X
$F_X(x)$	cumulative distribution function of X at x such that $P(X \leq x)$
$F_X^{-1}(y)$	inverse distribution function of X at y such that $y = F_X(F_X^{-1}(y))$
$\mu(y)$	estimate of inverse distribution function at quantile y
$\epsilon(y)$	error on inverse distribution estimate at quantile y
n	number of independently sampled Tor networks
$\hat{P}_i(X)$	probability distribution over X in network i
$\hat{F}_{X_i}(x)$	cumulative distribution function of X at x such that $\hat{P}_i(X \leq x)$
$\hat{F}_{X_i}^{-1}(y)$	inverse distribution function of X in network i at quantile y
$\hat{\mu}_i(y)$	estimate of inverse distribution function in network i at quantile y
$\hat{\epsilon}_i(y)$	error on inverse distribution estimate in network i at quantile y
m_i	number of simulations in sampled Tor network i
ν_{ij}	number of samples of X collected from sim j in net i
$\tilde{E}_{ij}(X)$	empirical distribution over ν_{ij} samples of X from sim j in net i
$\tilde{F}_{X_{ij}}(x)$	cumulative distribution function of X at x such that $\tilde{E}_{ij}(X \leq x)$
$\tilde{F}_{X_{ij}}^{-1}(y)$	inverse distribution function of X from sim j in net i at quantile y

guard relay selection). Multiple samples of each are needed to conduct a statistical inference and understand the error in these sampling processes.

We independently sample $n > 0$ Tor networks according to Section 3.3.2. The i th resulting Tor network is associated with a probability distribution $\hat{P}_i(X)$ which is specific to the i th network and the relays that were chosen when generating it. To estimate $\hat{P}_i(X)$, we run $m_i > 0$ simulations in the i th Tor network. During the j th simulation in the i th network, we sample ν_{ij} values of X from $\hat{P}_i(X)$ (e.g., we collect ν_{ij} time to last byte measurements from the simulation). These ν_{ij} samples form the empirical distribution $\tilde{E}_{ij}(X)$, and we have $\sum_{i=1}^n m_i$ such distributions in total (one for each simulation).

Estimating Distributions Once we have completed the simulations and collected the $\sum_{i=1}^n m_i$ empirical distributions, we then estimate the inverse distributions $\hat{F}_{X_i}^{-1}$ and F_X^{-1} associated with the sampled network and true probability distributions $\hat{P}_i(X)$ and $P(X)$, respectively.

First, we estimate each $\hat{F}_{X_i}^{-1}(y)$ at quantile y by taking the mean over the m_i empirical

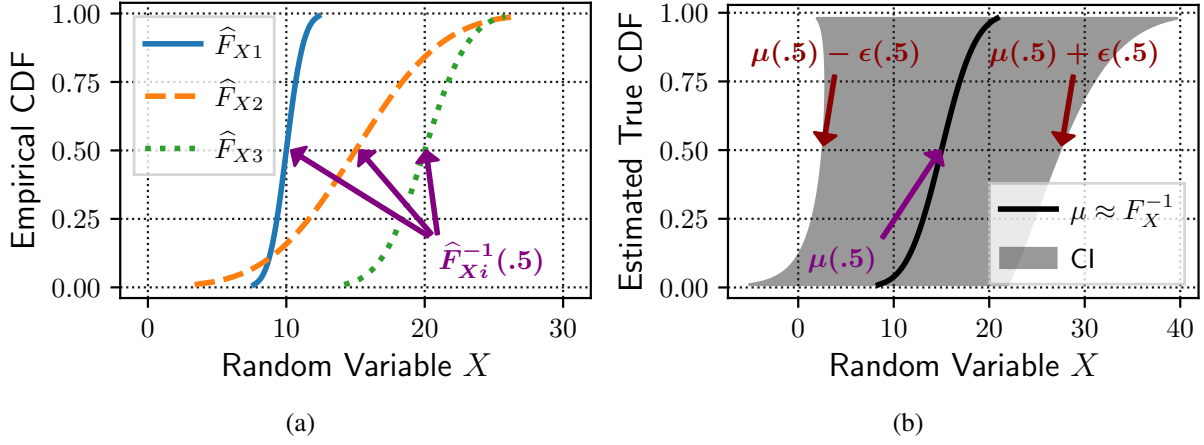


Figure 3.4: A synthetic example of estimating the cumulative distribution of a random variable X (e.g., time to last byte). (a) The mean in Equation 3.2 and standard deviation in Equation 3.5 are computed over the $n = 3$ values at each quantile. (b) The estimated true distribution from Equation 3.2 is shown with confidence intervals from Equation 3.7.

distributions from network i :

$$\hat{F}_{X_i}^{-1}(y) = \hat{\mu}_i(y) = \frac{1}{m_i} \sum_{j=1}^{m_i} \tilde{F}_{X_{ij}}^{-1}(y) \quad (3.1)$$

We refer to $\hat{\mu}_i$ as an estimator of $\hat{F}_{X_i}^{-1}$; when taken over a range of quantiles, it allows us to estimate the cumulative distribution $\hat{F}_{X_i}(x) = \hat{P}_i(X \leq x)$.

Second, we similarly estimate F_X^{-1} over all networks by taking the mean over the n distributions estimated above:

$$F_X^{-1}(y) \approx \mu(y) = \frac{1}{n} \sum_{i=1}^n \hat{\mu}_i(y) \quad (3.2)$$

We refer to μ as an estimator of F_X^{-1} ; when taken over a range of quantiles, it allows us to estimate the cumulative distribution $F_X(x) = P(X \leq x)$.

We visualize the process of estimating F_X^{-1} in Figure 3.4 using an example: Figure 3.4a shows $n = 3$ synthetic distributions where the upward arrows point to the $\hat{F}_{X_i}^{-1}$ values from network i at quantile $y = .5$, and Figure 3.4b shows the mean of those values as the estimator μ . The example applies analogously when estimating each $\hat{F}_{X_i}^{-1}$.

Computing Confidence Intervals We quantify the precision of our estimator μ using CIs. To compute the CIs, we first quantify the measurement error associated with the empirical samples. This will often be negligible, but a possible source of nontrivial measurement error is resolution

error; that is, if the empirical results are reported to a resolution of r (e.g., 0.01 s), the resolution error for each sample will be $\frac{r}{\sqrt{12}}$, and the resolution error ζ_i for the empirical mean $\hat{\mu}_i(y)$ of network i at quantile y is $\zeta_i = \frac{r}{\sqrt{12m_i}}$. Next, we quantify the sampling error associated with the estimates from Equation 3.1 and Equation 3.2. The error associated with $\hat{\mu}_i$ for network i at quantile y is:

$$\hat{\epsilon}_i(y) = \hat{\sigma}_i(y) \cdot t / \sqrt{m_i - 1} \quad (3.3)$$

where

$$\hat{\sigma}_i(y) = \sqrt{\frac{1}{m_i} \sum_{j=1}^{m_i} (\tilde{F}_{Xij}^{-1}(y) - \hat{\mu}_i(y))^2 + \zeta_i^2} \quad (3.4)$$

is the standard deviation over the m_i empirical values at quantile y (including the measurement error) and t is $t = T^{-1}(\frac{\alpha}{2} + 0.5, n - 1)$, i.e., the t -value from the Student's t -distribution at confidence level α with $m_i - 1$ degrees of freedom [Hoe71, §10.5.1]. This error $\hat{\epsilon}_i(y)$ accounts for the sampling error and estimated true variance of the underlying distribution at y . The error associated with μ at quantile y is:

$$\epsilon(y) = \delta(y) + \sigma(y) \cdot t / \sqrt{n - 1} \quad (3.5)$$

where

$$\sigma(y) = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{F}_{Xi}^{-1}(y) - \mu(y))^2} \quad (3.6)$$

is the standard deviation over the n estimated inverse distribution values at quantile y , and $\delta(y) = \frac{1}{n} \sum_{i=1}^n \hat{\epsilon}_i(y)$ is the mean error from $\hat{\mu}_i$ over all n sampled networks. $\epsilon(y)$ accounts for the sampling error introduced in the Tor network model generation and in the simulations. We can then define the CI at quantile y as the interval that contains the true value from the inverse distribution $F_X^{-1}(y)$ with probability α :

$$\mu(y) - \epsilon(y) \leq F_X^{-1}(y) \leq \mu(y) + \epsilon(y) \quad (3.7)$$

The width of the interval is $2 \cdot \epsilon(y)$, which we visualize at $y = .5$ with the downward arrows and over all quantiles with the shaded region in Figure 3.4b.

3.5.2 Discussion

Number of Samples Per Simulation Recall that we collect ν_{ij} empirical samples of the random variable X from simulation j in network i . If we increase ν_{ij} (e.g., by running the simulation for

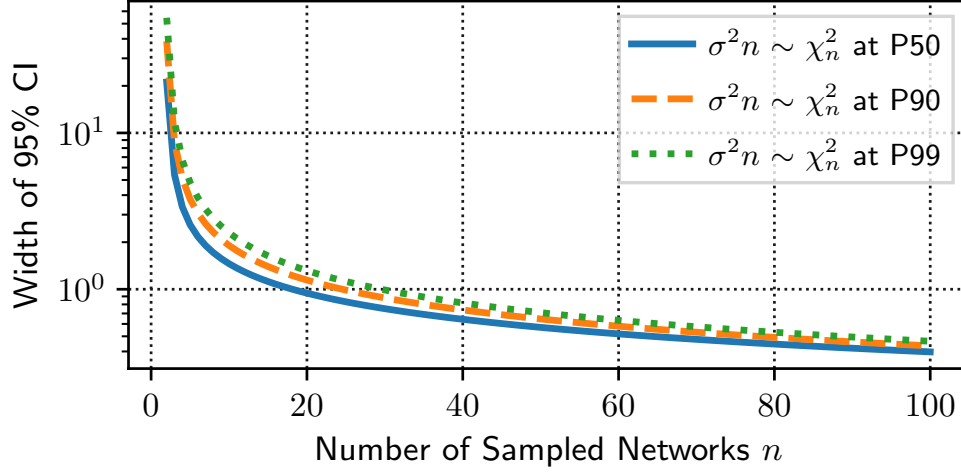


Figure 3.5: The width of the 95% CI when sampling a number of networks. The CI width (shown on the log-scale y-axis) can be significantly reduced by more than an order of magnitude after running experiments in fewer than 10 independently sampled Tor networks (when $\sigma^2 n$ is distributed according to χ_n^2).

a longer period of time), this will result in a “tighter” empirical distribution $\tilde{E}_{ij}(X)$ that will more closely resemble the probability distribution $\hat{P}_i(X)$. However, from Equation 3.1 we can see that $\tilde{E}_{ij}(X)$ only contributes a single value to the computation of $\hat{\mu}_i$ for each quantile. Therefore, once we have enough samples so that $\tilde{E}_{ij}(X)$ reasonably approximates $\hat{P}_i(X)$, it is more useful to run new simulations than to gather additional samples from the same simulation.

Number of Simulations Per Network Additional simulations in network i will provide us with additional empirical distributions $\tilde{E}_{i*}(X)$, which will enable us to obtain a better estimate of $\hat{P}_i(X)$. Moreover, it will also increase the precision of the CI by reducing $\hat{\epsilon}_i$ in Equation 3.3: increasing the number of $\tilde{E}_{i*}(X)$ values at each quantile will decrease the standard deviation $\hat{\sigma}_i$ (if the values are normally distributed) and the t -value (by increasing the number of degrees of freedom) while increasing the square root component (in the denominator of $\hat{\epsilon}_i$).

Number of Sampled Networks Additional simulations in independently sampled Tor networks will provide us with additional estimated $\hat{P}_i(X)$ distributions, which will enable us to obtain a better estimate of $P(X)$. Similarly as above, additional $\hat{P}_i(X)$ estimates will increase CI precision by reducing ϵ in Equation 3.5: the standard deviation σ and the t -value will decrease while the square root component will increase.

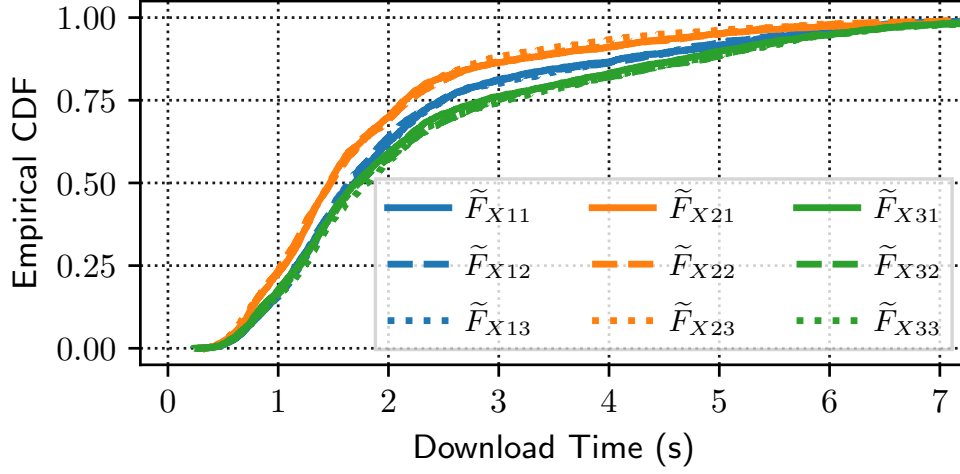


Figure 3.6: Sampling error introduced by Shadow is much less significant than error introduced by Tor network sampling (i.e., Section 3.3).

To give a concrete example, suppose σ^2 is distributed proportionally to a χ^2 distribution (the sum of the squares of n standard normal distributions, $\mathcal{N}(0, 1)$, is a χ_n^2 distribution, so if we assume the values of $\tilde{F}_{Xij}^{-1}(y) - \hat{\mu}_i(y)$ from Equation 3.6 are sampled from $\mathcal{N}(0, 1)$, then for any n networks, we would expect the value of $\sigma^2 n$ to be sampled from χ_n^2). The width of the resulting CI for each number of sampled networks $n \in [2, 100]$ at quantiles $y \in \{0.5, 0.9, 0.99\}$ (i.e., P50, P90, and P99, respectively) is shown in Figure 3.5. Notice that the y-axis is drawn at log-scale, and shows that the width of the CI can be significantly reduced by more than an order of magnitude after running experiments in even just a small number of sampled networks. Additionally, we can see that the main improvement in confidence results from the first ten or so sampled networks, after which we observe relatively diminishing returns.

Scale Another important factor to consider is the network scale $0 < s \leq 1$. Larger scales s (closer to 1) cause the probability distribution $\hat{P}_i(X)$ of each sampled network to cluster more closely around the true probability distribution $P(X)$, while smaller values cause the $\hat{P}_i(X)$ to vary more widely. Larger scales s therefore induce smaller values of $\sigma(y)$ and therefore $\epsilon(y)$. (See Section 3.6.3 for a demonstration of this phenomenon.)

Sampling Error in Shadow While ϵ includes the error due to sampling a scaled-down Tor network (i.e., Section 3.3), the main error that is accounted for in $\hat{\epsilon}_i$ is the sampling error introduced by the choices made in the simulator. If this error is low, running additional simulations in the

same network will have a reduced effect. To check the sampling error introduced by Shadow, we ran 9 simulations—3 simulations in each of 3 independently sampled networks of scale $s = 0.1$ —with unique simulator seeds. Figure 3.6 shows that the empirical distributions of the 50 KiB download times vary much more widely across sampled Tor networks than they do across simulations in the same network. Although it is in the abstract ideal to run multiple simulations in each of multiple sampled networks, our results indicate that it may be a better use of resources to run *every* simulation in an independently sampled network. We believe this to be a reasonable optimization if a lack of available computational resources is a concern.

Conclusions We have established a methodology for estimating the true distribution of random variables being studied across simulations in multiple Tor networks. Importantly, our methodology includes the computation of CIs that help researchers make statistical arguments about the conclusions they draw from Tor experiments. As we explained above and demonstrated in Figure 3.5, running simulations in smaller-scale Tor networks or in a smaller number of Tor networks for a particular configuration leads to larger CIs that limit us to drawing weaker conclusions from the results. Unfortunately, previous Tor research that utilizes Tor networks had focused exclusively on single Tor networks while completely ignoring CIs, rendering their results inconclusive (see Section 3.1.3). Our methodology has since supplanted the previous state-of-the-art methods [Aro24; DDB23; DSC+22; DT23; JW23; Mic21; OAD22; PAKE23; RE22; WHH22].

3.6 Case Study: Tor Usage and Performance

This section presents a case study on the effects of an increase in Tor usage on Tor client performance. Our primary goal is to demonstrate how to apply the methodologies we presented throughout this chapter through a concrete set of experiments.

3.6.1 Motivation and Overview

Growing the Tor network is desirable because it improves anonymity [ADS03] and access to information online [Uni20]. One strategy for facilitating wider adoption of Tor is to deploy it in more commonly used browsers. Brave now prominently advertises on its website Tor integration into its browser’s private browsing mode, giving users the option to open a privacy-enhanced tab that routes traffic through Tor [Bra24], and Mozilla has also expressed interest in providing a similar “Super Private Browsing” mode for Firefox users [Moz19]. However, Tor has never

been deployed at the scale of popular browser deployments (Firefox has over 180 million monthly active users [Moz24]), and many important research problems must be considered before such a deployment could occur [Gab19]. For example, deploying Tor more widely could add enough load to the network that it reduces performance to the extent that some users are dissuaded from using it [FGK+10] while reducing anonymity for those that remain [ADS03].

There has been little work in understanding the performance effects of increasing Tor network load as representative of the significant change in Tor usage that would likely occur in a wider deployment. Previous work that considered variable load did so primarily to showcase a new simulation tool [JH12] or to inform the design of a particular performance-enhancing algorithm [JSH12; JTG+18] rather than for the purpose of understanding network growth and scalability [KMG20]. Moreover, previous studies of the effects of load on performance lack analyses of the statistical significance of the reported results, raising questions as to their practical meaning.

Guided by the foundations that we set out in this chapter, we explore the performance effects of a sudden rise in Tor usage that could result from, e.g., a Mozilla deployment of Tor. In particular, we demonstrate the use of our methodologies with an example study of this simple hypothesis: *increasing the total user traffic load in Tor by 20% will reduce the performance of existing clients by increasing their download times and download error rates*. To study this hypothesis, we conduct a total of 420 simulations in independently sampled Tor networks across three network scale factors and two traffic load factors; we measure relevant performance properties and conduct a statistical analysis of the results following our methodology in Section 3.5. Our study demonstrates how to use our contributions to conduct statistically valid Tor performance research.

3.6.2 Experiment Setup

Experiments and Simulations We refer to an *experiment* as a unique pair of network scale s and load ℓ configurations, and a *simulation* as a particular execution of an experiment configuration. We study our hypothesis with a set of 6 experiments; for each experiment, we run multiple simulations in independent Tor networks so that we can quantify the statistical significance of the results following our guidance from Section 3.5.

Tor Network Scale and Load The Tor network scales that a researcher can consider are typically dependent on the amount of RAM to which they have access. Although we were able to run a 100% Tor network for our evaluation in Section 3.4, we do not expect that access to a machine with 4 TiB of RAM, as was required to run the simulation, will be common (and, as

Table 3.6: Tor usage and performance experiments in Shadow

Scale s	Load ℓ	Sims n	CPU*	RAM/Sim [†]	Run Time/Sim [‡]
1%	100%	100	4×8	35 GiB	4.8 hours
1%	120%	100	4×8	50 GiB	6.7 hours
10%	100%	100	4×8	355 GiB	19.4 hours
10%	120%	100	4×8	416 GiB	23.4 hours
30%	100%	10	8×8	1.07 TiB	4 days, 21 hours
30%	120%	10	8×8	1.25 TiB	5 days, 22 hours

* 4×8-core Intel Xeon E5 @3.3 GHz; 8×8-core Intel Xeon E5 @2.7 GHz.

[†] The median of the per-simulation max RAM usage over all simulations.

[‡] The median of the per-simulation run time over all simulations.

mentioned, is no longer possible with current versions of Shadow). We therefore focus our study on multiple smaller network scales with more accessible resource requirements while showing the change in confidence that results from running networks of different scales. In particular, our study considers Tor network scales of 1%, 10%, and 30% ($s \in \{0.01, 0.1, 0.3\}$) of the size of the true Tor network. At each of these network scales, we study the performance effects of 100% and 120% traffic load ($\ell \in \{1.0, 1.2\}$) using a process scale factor of $p = 0.01$, i.e., each TGen process simulates $1/0.01 = 100$ Tor users.

Number of Simulations Another important consideration in our evaluation is the number n of simulations to run for each experiment. As explained in Section 3.5, running too few simulations will result in wider confidence intervals that will limit us to weaker conclusions. The number n of simulations that should be run typically depends on the results and the arguments being made, but in our case we run more than we require to validate our hypothesis in order to demonstrate the effects of varying n . As shown in the left part of Table 3.6, we run a total of 420 simulations across our 6 experiments (three network scales and two load factors) using two machine profiles: one profile included 4×8-core Intel Xeon E5-4627 CPUs running at a max clock speed of 3.3 GHz and 1.25 TiB of RAM; the other included 8×8-core Intel Xeon E5-4650 CPUs running at a max clock speed of 2.7 GHz and 1.5 TiB of RAM.

Simulation Configuration We run each simulation using an independently sampled Tor network in order to ensure that we produce informative samples following our guidance from Section 3.5.

Table 3.7: Network composition in each simulation

Scale s	Relays	DirAuth	Guard	Middle	Exit	E+G [†]	Markov	Perf [‡]	Server
1%	67	3	20	36	4	4	100	8	10
10%	652	3	204	361	40	44	792	79	79
30%	1,948	3	612	1,086	118	129	2,376	238	238

[†] **E+G**: Relays with both the exit and guard flags

[‡] **Perf**: Benchmark clients

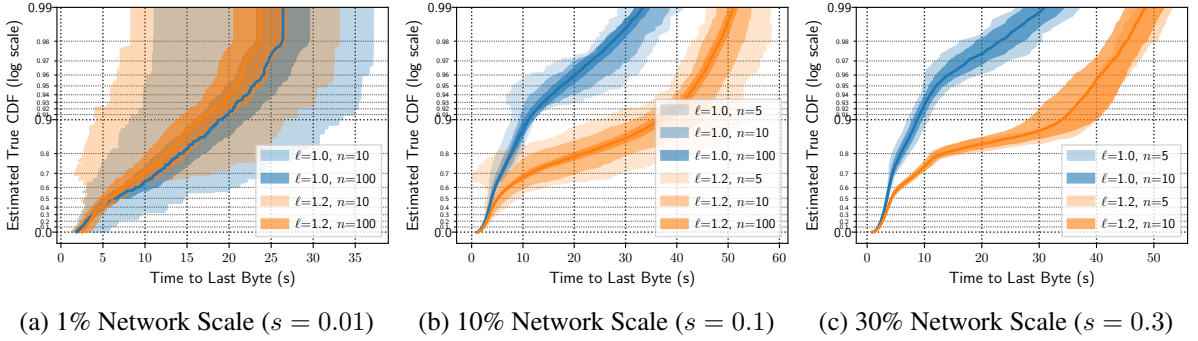


Figure 3.7: Time to last byte in seconds of 1 MiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s . The results from each experiment are aggregated from n simulations following Section 3.5, and the CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance.

Each Tor network is generated following our methodology from Section 3.3 using the parameter values described above and Tor network state files from January 2019. The resulting network composition for each scale s is shown in Table 3.7.

Each simulation was configured to run for 1 simulated hour—the default experiment duration used historically, as a somewhat arbitrary middle ground between gathering enough data in an experiment and run time, with the first 20 minutes dedicated to achieving steady state and discarded from reported results. The relays bootstrapped a Tor overlay network within the first 5 minutes; all of the TGen clients and servers started their traffic generation process within 10 simulated minutes of the start of each simulation. TGen streams created by Markov clients were set to time out if no bytes were transferred in any contiguous 5 simulated minute period (the default apache client timeout), or if the streams were not complete within an absolute time of 10 simulated minutes. Timeouts for streams created by benchmarking clients were set to 15, 60, and

120 seconds for 50 KiB, 1 MiB, and 5 MiB transfers, respectively.

3.6.3 Results

During each simulation, we measure and collect the properties that allow us to understand our hypothesis. Ultimately, we would like to test if increasing the traffic load on the network by 20% (from $\ell = 1.0$ to $\ell = 1.2$) will reduce client performance. Therefore, we focus this study on client *download time* and *download error rates* while noting that it will very likely be useful to consider additional properties when studying more complex hypotheses (see Section 3.2).

For each experiment, we combine the results from the n simulations¹⁶ following the methodology outlined in Section 3.5 and present the estimated true cumulative distributions with the associated CIs (as in Figure 3.4) at $\alpha = 95\%$ confidence. We plot the results for varying values of n as overlapping intervals (the CIs tighten as n increases) for instructional purposes. Finally, we compare our results across network scales s to highlight the effect of scale on the confidence in the results.

Client Download Time The time it takes to download a certain number of bytes through Tor (i.e., the time to first/last byte) allows us to assess and compare the overall performance that a Tor client experiences. We measure download times for the performance benchmarking clients throughout the simulations. We present in Figure 3.7 the time to last byte for 1 MiB file downloads, while noting that we find similar trends for other file download sizes as shown in Figures A.1, A.2, and A.3 in Appendix A. The CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance as is typically used as an indication of usability.

Figure 3.7a shows the result of our statistical analysis from Section 3.5 when using a network scale of 1% ($s = 0.01$). Against our expectation, our estimates of the true CDFs (i.e., the solid lines) indicate that the time to download 1 MiB files actually *decreased* after we increased the traffic load by 20%. However, notice the extent to which the confidence intervals overlap: for example, the width of the region of overlap of the $\ell = 1.0$ and $\ell = 1.2$ CIs is about 20 seconds at P90 (i.e., at $x \in [8, 28]$ seconds) when $n = 10$, and is about 3 seconds at P90 (i.e., at $x \in [16.5, 19.5]$ seconds) when $n = 100$. Importantly, the estimated true CDF for $\ell = 1.0$ falls completely within the CIs for $\ell = 1.2$ and the estimated true CDF for $\ell = 1.2$ falls completely within the CIs for $\ell = 1.0$, even when considering $n = 100$ simulations for each experiment. Therefore, it is possible that the x position of the true CDFs could actually be swapped compared

¹⁶We ignore the results from the first 20 simulated minutes of each simulation to allow time for the network to bootstrap and reach a steady state.

to what is shown in Figure 3.7a. If we had followed previous work and ignored the CIs, it would have been very difficult to notice this statistical possibility. Based on these results alone, we are unable to draw conclusions about our hypothesis at the desired confidence.

Our experiments with the network scale of 10% offer more reliable results. Figure 3.7b shows the extent to which the CIs become narrower as n increases from 5 to 10 to 100. Although there is some overlap in the $\ell = 1.0$ and $\ell = 1.2$ CIs at some $y < 0.9$ values when n is either 5 or 10, we can confidently confirm our hypothesis when $n = 100$ because the estimated true CDFs and their CIs are completely distinguishable. Notice that the CI precision at $n = 10$ and $n = 100$ has increased compared to those from Figure 3.7a, because the larger scale network produces more representative empirical samples.

Finally, the results from our experiments with the network scale of 30% reinforce our previous conclusions about our hypothesis. Figure 3.7c shows that the estimated true CDFs and their CIs are completely distinguishable, allowing us to confirm our hypothesis even when $n = 5$. However, we notice an interesting phenomenon with the $\ell = 1.2$ CIs: the CI for $n = 10$ is unexpectedly *wider* than the CI for $n = 5$. This can be explained by the analysis shown in Figure 3.5: as n approaches 1, the uncertainty in the width of the CI grows rapidly. In our case, the empirical distributions from the first $n = 5$ networks that we generated happened to be more closely clustered by chance, but $n = 10$ resulted in a more diverse set of sampled networks that produced more varied empirical distributions. Our conclusions happen to be the same both when $n = 5$ and when $n = 10$, but this may not always be the case (e.g., when the performance differences between two experiments are less pronounced). We offer general conclusions based on our results later in this section.

Client Download Error Rate The client download error rate (i.e., the fraction of failed over attempted downloads) helps us understand how additional traffic load would impact usability. Larger error rates indicate a more congested network and represent a poorer user experience. We measure the number of attempted and failed downloads throughout the simulations, and compute the download error rate across all downloads (independent of file size) for each performance benchmarking client. We present in Figure 3.8 the download error rate across all benchmarking clients. (Note that another general network assessment metric, Tor network goodput, is shown in Figure A.4 in Appendix A.)

Figure 3.8a shows the result of our statistical analysis from Section 3.5 when using a network scale of 1% ($s = 0.01$). As with the client download time metric, we see overlap in the $\ell = 1.0$ and $\ell = 1.2$ CIs when $n = 10$. Although it appears that the download error rates *decrease* when adding 20% load (because the range of the $\ell = 1.0$ CI is generally to the right of the range of the $\ell = 1.2$ CI), we are unable to draw conclusions at the desired confidence when $n = 10$. However, the $\ell = 1.0$ and $\ell = 1.2$ CIs become significantly narrower (and no longer overlap) with $n = 100$

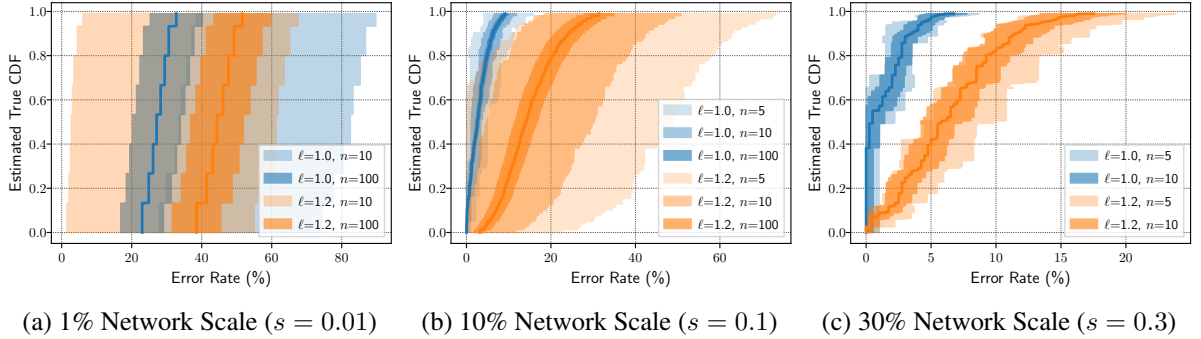


Figure 3.8: The download error rate for downloads of all sizes from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s . (I.e., the fraction of failed over attempted downloads.) The results from each experiment are aggregated from n simulations following Section 3.5.

simulations, and it becomes clear that adding 20% load increases the error rate.

Our experiments with the network scale of 10% again offer more reliable results. Figure 3.8b shows significant overlap in the $\ell = 1.0$ and $\ell = 1.2$ CIs when $n = 5$ simulations and a very slight overlap in CIs when $n = 10$. However, based on the estimated true CDF and CIs when $n = 100$, we can again confidently conclude that increasing the traffic load by 20% increases the download error rate because the CIs are clearly distinguishable. Notice that the CI precision for $\ell = 1.2$ compared to the CI precision for $\ell = 1.0$ offers an additional insight into the results: the error rate is more highly varied when $\ell = 1.2$, indicating that the user experience is much less *consistent* than it is when $\ell = 1.0$.

Finally, the results from our experiments with the network scale of 30% again reinforce our previous conclusions about our hypothesis. Figure 3.8c shows that the estimated true CDFs and their CIs are completely distinguishable, allowing us to confirm our hypothesis even when $n = 5$.

Conclusions We offer some general observations based on the results of our case study. First, our results indicate that it is possible to come to similar conclusions by running experiments in networks of different scales. Generally, fewer simulations will be required to achieve a particular CI precision in networks of larger scale than in networks of smaller scale. The network scale that is appropriate and the precision that is needed will vary and depend heavily on the experiments and metrics being compared and the hypothesis being tested. However, based on our results, we suggest that networks at a scale of at least 10% ($s \geq 0.1$) are used whenever possible, and we strongly recommend that 1% networks be avoided due to the unreliability of the results they

generate. Second, some of our results exhibited the phenomenon that increasing the number of simulations n also decreased the CI precision, although the opposite is expected. This behavior is due to random sampling and is more likely to be exhibited for smaller n . Along with the analysis from Section 3.5, our results lead us to recommend that no fewer than $n = 10$ simulations be run for any experiment, independent of the network scale s .

3.7 Conclusion

In this chapter, we developed foundations upon which future Tor performance research could, and ultimately did, build. The foundations we developed include: (i) a new Tor experimentation ontology; (ii) a new Tor network modeling methodology and supporting tools that produce *more representative* Tor networks (Section 3.3); (iii) accuracy and performance improvements to the Shadow simulator that allow us to run Tor simulations *faster* and at a *larger scale* than was previously possible (Section 3.4); and (iv) a methodology for conducting statistical inference of results generated in scaled-down (sampled) Tor networks (Section 3.5). We showcased our modeling and simulation scalability improvements by running simulations with 6,489 relays and 792k users, the largest known Tor network simulations and the first at a network scale of 100% (Section 3.4.3), and demonstrating a measurable improvement in accuracy over the previous state of the art. Building upon the above foundations, we conduct a case study of the effects of traffic load on client performance in the Tor network through a total of 420 Tor simulations across three network scale factors and two traffic load factors (Section 3.6). Our case study demonstrates how to apply our methodologies for modeling Tor networks and for conducting sound statistical inferences of results.

Conclusions We find that: (i) significant reductions in RAM are possible by representing multiple Tor users in each Tor client process (Section 3.4.3); (ii) it is feasible to run 100% Tor network simulations on high-memory servers in a reasonable time (less than 2 weeks) (Section 3.4.3); (iii) running multiple simulations in independent Tor networks is necessary to draw statistically significant conclusions (Section 3.5); and (iv) fewer simulations are generally needed to achieve a desired CI precision in networks of larger scale than in those of smaller scale (Section 3.6).

Limitations and Future Work Although routers in Shadow drop packets when congested (using CoDel), we describe in Section 3.3.1 that we do not model any additional artificial packet loss. However, it is possible that packet loss or corruption rates are higher in Tor than in Shadow

(e.g., for mobile clients that are wirelessly connected, and clients connecting from China), and modeling this loss could improve realism. Future work should consider developing a more realistic packet loss model that is, for example, based on measurements of actual Tor clients and relays.

In Section 3.3.2 we describe that we compute some relay characteristics (e.g., consensus weight, bandwidth rate and burst, location) using the median value of those observed across all consensus and server descriptors from the staging period. Similarly, in Section 3.3.2 we describe that we select m relays from those available by “bucketing” them and choosing the relay with the median bandwidth capacity from each bucket. These selection criteria may not capture the full variance in the relay characteristics. Future work might consider alternative selection strategies—such as randomly sampling the full observed distribution of each characteristic, choosing based on occurrence count, or choosing uniformly at random—and evaluate how such choices affect simulation accuracy.

Our traffic modeling approach in Section 3.3.2 allows us to reduce the RAM required to run simulations by simulating $1/p$ users in each Tor client process. This optimization yields the following implications. First, we disable guard relays in our model because Tor does not currently support multiple guard “sessions” on a given Tor client. Future work should consider either implementing support for guard relay “sessions” in the Tor client, or otherwise managing guard relay selection and circuit assignment through the Tor control port. Second, simulating $1/p$ users on a Tor client results in “clustering” these users in the city that was assigned to the client, resulting in lower location diversity. Choosing values of p closer to 1 would reduce this effect. Third, setting $p < 1$ reduces the total number of Tor clients and therefore the total number of network consensus fetches. Because these fetches occur infrequently in Tor, the network impact is negligible relative to the total amount of traffic being generated by each client.

Finally, future work might consider sampling the Tor network at scales $s > 1.0$, which could help us better understand how Tor might handle growth as it becomes more popular.

Chapter 4

Using Tor to Protect Metadata in Messengers

Now that we are equipped with tools for performing valid experiments involving Tor, we can focus on techniques for using Tor. In this chapter, we examine using Tor in combination with messaging applications (messengers) to provide metadata protection, using (in part) the methodology developed in Chapter 3.

In the past decade, messengers that market themselves as providing end-to-end encryption (e2ee) have proliferated. While deploying e2ee is laudable, both from a privacy and a security perspective, it is not the end of the story. Correctly implemented e2ee can protect the contents of messages and files relayed by such applications, but it does not provide any guarantees of protection of metadata—i.e., data about the use of the application, such as who is communicating with whom, when the application is or has been in use, or whether a particular individual is using the application at all. Often, metadata is of more relevance to adversaries than the content, as has been pointed out many times using the now-infamous quote from former NSA and CIA director General Michael Hayden: “We kill people based on metadata.” [GHC14]

This then raises the obvious question of how the makers of these applications have responded to the threat of metadata collection. The answer appears to fall into three categories: **ignore existence** of the threat, **deflect responsibility** away from the application, or **integrate solutions** into the application itself. In this chapter, we explore these categories, and how they demonstrate the importance of facilitating simple techniques for integrating metadata protection. Towards that end, we investigated to what degree Tor, which has been widely recommended in this space [Esr24; Mar19; MKA+21; Mol23], can be used to simplify metadata protection in these respective types of applications. In our investigation, we found that simply using Tor on existing messengers that

were never built with the express intent of metadata protection is insufficient to provide much, if any, additional privacy, and that if application developers wish to allow their users to make decisions about the protection of their metadata, conscious design with that use in mind must be employed—much as is the case with other kinds of applications [Fle09; MAB+10]. As an example, we look at the metadata leakage of Signal in particular, and find despite claims that Tor can be used for network anonymity in Signal, little actual protection would be gained by doing so. We then look at the practicality of large-scale deployment of messaging applications that use Tor—both as a simple wrapper for current popular messaging apps, and for applications that integrate Tor into their design. We find that while there are no immediate performance problems with a moderate number of users using Tor for their communications, greatly scaling applications based on onion services could adversely affect some types of Tor traffic.

In summary, we make the following contributions in this chapter: (i) we categorize and analyze the existing responses to threats of metadata leakage among popular e2ee messengers; (ii) we examine the metadata implications of using Tor in combination with messengers, and show that the advantages of proxying Signal over Tor are negligible; (iii) we perform a series of network simulations to show the impact wide deployment of Tor on messengers would have on said messengers and the Tor network; and (iv) we provide a wide suite of tools for further research into the analysis of messenger network traffic, both in simulations and the real world.¹

4.1 End-to-End Encrypted Messengers

With one in four people globally using WhatsApp every month alone [Dix24], most are likely to at least be aware of the existence of e2ee messengers. Tracing their lineage to the instant messaging applications of the 1990s, where security was generally an afterthought, current e2ee messengers have adapted to more recent network conditions, form factors, and threat models. Originally, securing the contents of early messengers from the servers transmitting them required additional applications or plugins—likely the “Off-the-Record messaging” (OTR) implementation [BGB04]. OTR was used as the basis of encryption in the TextSecure app (itself a means of layering encryption on SMS cell phone messages), then modified as the app matured, eventually becoming the Signal app and Signal Protocol. It became progressively more common for e2ee to be expected as part of basic security in messengers, with the publicly available and thoroughly audited Signal Protocol becoming the most common basis for encryption protocols among e2ee messenger apps (including in WhatsApp and the newly deployed e2ee in Facebook messages).

As part of this transition, applications that work well in the context of smartphones also

¹<https://git-crysp.uwaterloo.ca/j3tracey/MGen>

became expected. While some synchronous messengers do still exist, they face a higher barrier to adoption in a world where connectivity can drop for extended periods without warning. With competition between features drawing users towards or away from each platform, the functionality also expanded over time, adopting features such as delivery and read receipts, typing indicators, voice and video calls, and larger groups.

4.2 Solutions to Protect Metadata

In the space of secure messengers, metadata is rarely as rigorously modeled or documented as the encryption in use. While some will advertise their ability to prevent particular attacks targeting metadata, there is little in the way of a systematic understanding of what even is worth protecting from whom, let alone a means of comparing these tools. Of course, previous work exists in the academic and anonymity network space [KBS+19; SG24], but little of this translates directly into how these messengers operate in practice. To provide some framing and orientation into real-world policy and design around metadata, we provide here some high-level observations on the way the messengers and the organizations that create them treat threats to user metadata.

4.2.1 Ignore Existence

The first response to the problem of metadata leakage is to ignore the existence of any problem. That is, many encrypted messaging applications do not address metadata leakage as a threat at all (much as how many messaging applications and services disputed the need for concern with encryption ten years ago). “Ignore” here may give incorrect connotations, in that the threat may be actively interrogated, but nevertheless left unaddressed. In reality, this category contains many justifications, with the core unifying element of them being how the application provider manifests its metadata protection policy—i.e., there is none.

Note that, similar to encryption in secure messengers, we frame metadata protection with the server operator (where one exists) as some form of adversary. Just as a typical analysis would not consider messages whose contents are analyzed on a service’s internal servers to be secure (irrespective of any protections that are intended to prevent those contents from leaving their servers), we consider any metadata leakage to the service itself to be a threat, regardless of the protections the service takes to ensure only it has access to said metadata.

WhatsApp, for example, is the most popular e2ee messenger, but has never made any statements towards protecting metadata from its own operations. In fact, parent company Meta (*née* Facebook)’s then-COO Sheryl Sandberg has made comments towards the exact opposite, stating

one of the reasons for WhatsApp’s encryption is, ironically, to ensure they retain the ability to record metadata [You17]:

The goal for governments is to get as much information as possible. And so when there are message services like WhatsApp that are encrypted, the message itself is encrypted but the metadata is not, meaning that you send me a message, we don’t know what that message says but we know you contacted me. If people move off those encrypted services to go to encrypted services in countries that won’t share the metadata, the government actually has less information, not more.

In general, Meta’s policies around metadata collection in WhatsApp have been self-contradictory, and difficult to parse and interpret [Bid24; EGS21; Met23; Moz21; Wha21a; Wha21b]. Regardless, we do know that WhatsApp has indicated its willingness to collect metadata, and that it has a financial interest in doing so. This implies that an active adversary, who is willing to modify its behavior to better position itself to collect this metadata, is a better model than a passive adversary opportunistically gathering information without detection for WhatsApp in particular. Therefore, any technique for additional metadata protection for WhatsApp end users should anticipate this form of adversary.

Other messengers that fall into this category include Meta’s other messenger, called Messenger, which recently launched e2ee by default [Cri23], and Apple’s iMessage, for which there is little public information available, though based on law enforcement records, it appears they store some amount of metadata [Bid16].

4.2.2 Deflect Responsibility

The second category of responses to the matter of metadata is to deflect (or offset) responsibility by stating it is a problem orthogonal to what the application is attempting to solve. That is, the application providers claim to allow for some other mechanism of protecting metadata to be used in conjunction with the application they provide, but it is not their application that is providing that protection. While WhatsApp, which uses a protocol based on the Signal protocol, is in the previous category, the Signal messenger itself falls into this category.

The Signal developers have taken steps to reduce the metadata footprint at-rest on their servers by expressly choosing not to record such information,² and added features to mitigate the threat—most notably, features Signal calls Private Contact Discovery, the Signal Private Group System, and Sealed Sender. For Signal, Private Contact Discovery consists of using a hash table in an Intel

²<https://signal.org/bigbrother/>

Software Guard Extensions (SGX) enclave with a hash of the phone number as the key when users query for known contacts, so that it would be more difficult for Signal to observe the user’s contact list [Mar17].³ The Signal Private Group System refers to the infrastructure in place to keep group chat configuration (e.g., membership) end-to-end encrypted [CPZ20; OLe19].

Most relevant for our work, Sealed Sender refers to a feature Signal uses to minimize the amount of data recorded on who is sending Signal messages [Lun18]. Prior to its deployment in 2018, all Signal messages were sent via authenticated channels with the Signal server to provide assurance to recipients of the identity of the sender, and to allow for rate limiting and reduce abuse of the service. Sealed sender modifies this in two ways, so that the sender is no longer visible to the server. First, sender certificates (which are used to identify senders to recipients) are sent end-to-end encrypted. Second, to send a “sealed sender” message, the sending client must prove knowledge of the recipient’s “delivery token”, which is derived from the recipient’s profile key—the assumption being that rate limiting and abuse protection will not be necessary if the sender has already received that recipient’s profile (the sender is in the recipient’s contacts, the recipient has sent a message to the sender, or the recipient has explicitly approved giving the sender their profile—and the sender is not blocked). As the name implies, sealed sender only conceals the sender of the message from the server—the recipient is still visible, so that the server knows who to relay the message to.

While these features are laudable, they do not provide complete metadata protection. For example, all of these features make no attempt at addressing the privacy properties of the network protocols involved. More generally, these features are motivated by the ability for a trusted server to record and store as little plaintext information as possible [Mar19]—a “sealed sender” does not address the fact that the server must necessarily see the sender’s IP address, for example. However, when questioned on the utility of these measures when assuming a malicious server, as is standard with regards to content protection, the response from Signal is a common refrain from this category of responses: users who wish to protect that information can use Tor [Mar19]. Such an answer is emblematic of this category of responses—the application developers will take measures to not record metadata, but leave responsibility for robust protection with the user. Other examples of messengers in this category would be some deployments of the Matrix protocol,⁴ which generally stores more metadata than Signal, but can use self-hosted servers.

³<https://github.com/signalapp/ContactDiscoveryService-Icelake>

⁴<https://matrix.org>

4.2.3 Integrate Solutions

The final category of observed responses to metadata leakage are attempts to protect it in the design of the application itself. Of course, designs have been proposed in this category since long before the proliferation of end-to-end encryption—e.g., the original mixnet designs were created for exactly this metadata leakage. Yet none of these designs have seen widespread adoption, generally as a result of poor user experience. More recent designs hope to address this lack of adoption by building on the existing metadata protection provided by Tor. Specifically, in contrast to the previous category’s reliance on Tor as a simple proxy, applications in the vein of responses in this category integrate the design of the application with Tor’s, so that they may rely on a wider range of its functionality and account for its pitfalls. (Note that while we focus on Tor in this thesis, other technologies that provide network-layer protection could ostensibly be used instead, such as i2p⁵ or Nym [DHK21], with potentially different metadata protection properties with different trade-offs than those presented in Chapter 2.)

The first messaging application in this category was the (now abandoned) Ricochet.⁶ The primary insight of Ricochet was that many of the properties sought after in a metadata-protecting messenger can be achieved via onion services. Ricochet therefore opted to run the messaging client as an onion service, with identifiers for users being (slightly modified) onion addresses; i.e., if Alice wished to contact Bob on Ricochet, then Alice would need Bob’s Ricochet ID, consisting of a non-human readable string of text. As a result, there was no central server with Ricochet chat connections. Instead, Ricochet clients connected using the beneficial onion service properties described in Chapter 2. Furthermore, each connection is established over its own (pair of) circuits, each conceptually indistinguishable from any other. Finally, as with all onion services, the identifiers are self-authenticating—so long as users are confident that the identifiers used do correspond to the person they are trying to message, the messages will be securely sent and authenticated.

There are some drawbacks to this approach, however. For one, Ricochet’s design does not permit asynchronous messaging. Because each user is running the server that peers connect to as part of their Ricochet client, once the client is shut down or disconnects, messages will fail to be delivered to that user. As previously mentioned, this choice places a barrier for Ricochet adoption. Additionally, Ricochet does not provide any means of group messaging.

Both of these problems are in the process of being addressed by new systems under development, such as Cwtch [Lew18]⁷ and Quiet.⁸ Cwtch builds on Ricochet, with the initial handshakes

⁵<https://geti2p.net>

⁶<https://web.archive.org/web/20231215140743/https://ricochet.im/>

⁷<https://cwtch.im>

⁸<https://tryquiet.org>

between clients performed via minor adjustments on Ricochet’s protocol. For most conversations, Cwtch continues to use this connection, with most of its advantages over Ricochet consisting of an improved user experience. In the case of (currently experimental) asynchronous messages and group chats, this handshake is instead used to select a Cwtch server. Each party in the group establishes a Ricochet connection with the Cwtch server, using new ephemeral Ricochet-style IDs. This server relays all messages it receives to all peers (i.e., Cwtch clients) that have connected to it. Whenever a peer wishes to send a message to a group, it encrypts it to the intended recipients, sends it to the server, which then broadcasts it to all peers, even those it was not intended for. Peers who are not part of the group will quickly fail to decrypt the message, while intended recipients will successfully do so. Because of the protections provided by onion services, as well as the broadcast mechanism, the server can provide asynchronous message retrieval and group chats, but will never learn who is a member of which groups, nor who is communicating with whom. Proposed improvements to this system increase efficiency by relying on fuzzy message detection [BLMG21; Ope21]. Quiet, on the other hand, attempts to address these problems by layering an IPFS [Ben14] distributed data store among the group members (again, connected via onion services).

One other possible example in this category is Briar, which also uses onion services, but only when the sender and recipient devices are not on the same local network (e.g., WiFi or Bluetooth).⁹

4.3 Analysis of Metadata Protection

In this section, we detail our analysis of the metadata protection provided by the WhatsApp and Signal messengers, in their representative role of the ignore and deflect categories, respectively. Specifically, we will examine what sort of metadata is available to a malicious server when all connections are made through Tor circuits. In this thesis, we will exclude the possibility of malicious clients as a threat.

4.3.1 Methodology

While in the case of Signal we have the advantage of open client and server source code, this is not generally the case for all apps worthy of study. Furthermore, even in the case of Signal, there are no known alternative deployments of the Signal server, and its alleged code has been observed to drift from its known deployed behavior at times [Von21]. Even for the Signal client, while we could read the source code or modify it to intercept traffic, we can avoid errors in interpretation or

⁹<https://briarproject.org>

introducing bugs by intercepting the TLS method calls from the Android app—a technique that should generalize to most apps.

Towards that end, we created a toolkit for recording traffic between a messenger client and its server. The toolkit consists of several off-the-shelf tools, some modified tools, and our own scripts, all of which are made publicly available.¹⁰

Our setup uses an Android emulator image that ships with the official Android SDK. By using the emulator image, our setup does not require a physical rooted device to intercept TLS traffic as plaintext. We do still need root access, however, as well as a mechanism of modifying the system, which we do using the well-known Magisk app.¹¹ However, Magisk does not work with emulator images by default, so we have forked, revived, and improved a collection of scripts to patch the emulator image and allow Magisk to function properly.

For a fully featured shell, we install the Termux app,¹² and use its package manager to install various useful utilities, including tcpdump¹³ and mitmproxy,¹⁴ which can be used to locally record all network traffic and perform machine-in-the-middle (MITM) attacks on said traffic. To hook into the TLS methods and modify them when executed by apps, we use the Frida instrumentation toolkit,¹⁵ and a script to interface it with Magisk for integration into the running Android system. Using our scripts to manage the above tools, as well as interactions with Signal (or the app in question), a packet capture file and key file are produced. By using a plugin we wrote for analyzing Signal packet captures, these files—or, with a slight modification to the steps, live network data—can be used with Wireshark to view traffic from the vantage of the Signal server.

4.3.2 WhatsApp

As the most popular messenger by a wide margin, much of our data comes from WhatsApp—including information about message sizes, timings, and distribution of group sizes, as described in Section 4.4.2. However, as WhatsApp is a closed-source application, our analysis of network and application metadata leakage was somewhat stymied. We attempted to analyze the network traffic of WhatsApp messages by intercepting our own communications, using the toolkit described above. In the process of investigating WhatsApp’s multiple, non-standard TLS libraries (causing some faulty, terminated TLS connections in the process), the main account we used for the

¹⁰<https://git-crysp.uwaterloo.ca/j3tracey>

¹¹<https://github.com/topjohnwu/Magisk>

¹²<https://termux.dev>

¹³<https://www.tcpdump.org>

¹⁴<https://mitmproxy.org>

¹⁵<https://frida.re>

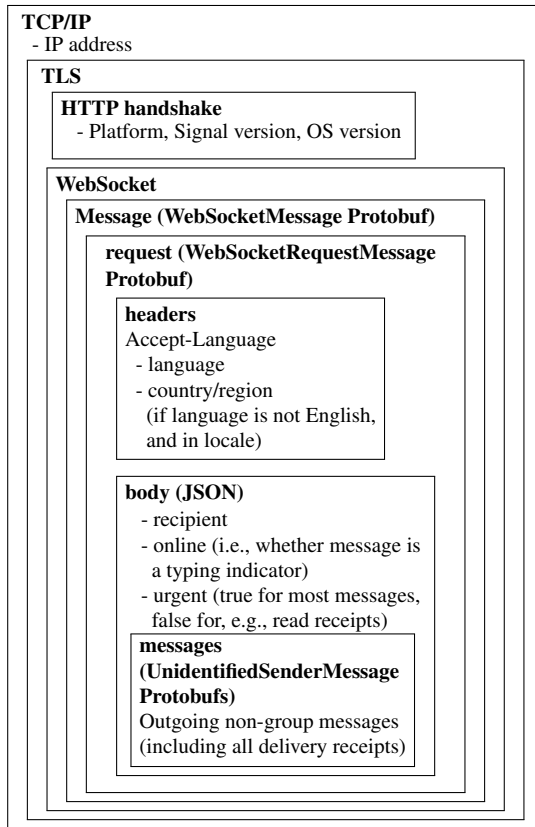
investigation was quickly banned. We believe our activities were well within the Meta bug bounty terms that WhatsApp uses,¹⁶ and reached out to WhatsApp for clarification on what was unacceptable and what we could do to continue our investigation. We were only told that they “received a large number of complaints about [our] account and to protect [their] users’ privacy, [they] won’t disclose the nature of the complaints”, and that they would not communicate with us further. Given that the only activity on the account was joining large public group chats without sending messages (see Section 4.4.2), and sending private messages to our other accounts, we do not know what was meant by this explanation. Regardless, while we could have continued with our other accounts that had not yet been banned, doing so would violate their terms of service with respect to circumventing bans, so we ceased all research involving further use of the WhatsApp client (though we still do make use of data collected to that point). With regards to metadata leakage from WhatsApp over Tor, all we can say is that it almost certainly leaks at least as much as Signal.

4.3.3 Signal

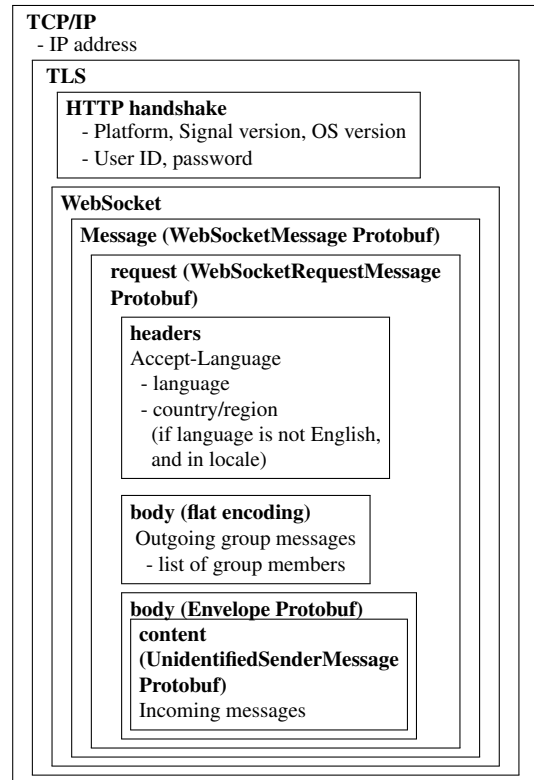
Unlike WhatsApp, we were able to successfully intercept our own traffic from the Signal app without difficulties from Signal. While the details can vary depending on the type of message and user configuration, we found that Signal messages rely on a deeply nested set of protocols, with full functionality of the app making use of several servers. The connections that make up a typical Signal session from an Android client, annotated with notable data associated with that layer, can be found in Figure 4.1. There is (1) a channel where the client provides no login credentials, where outgoing sealed sender messages to individual recipients (including delivery receipts) are sent, (2) a channel where the client provides a user ID and password (both random strings generated on signup, never displayed to the user) to primarily receive all messages, and (3,4) channels for uploading and downloading file attachments. Additional channels exist for other services as well, such as managing public keys stored server-side, but are not documented here as they are not relevant for our purposes. (We also do not account for information leaked via Google’s Play Services—most notably, push notifications—since, while presumably rare, users can use Signal on Android devices without these.) Interestingly, it appears that outgoing group messages (which use a different structure from normal dyadic messages to reduce redundant traffic, though still a form of sealed sender) are always sent on the authenticated message channel, and not on the unauthenticated channel used for most outgoing messages.

¹⁶<https://www.facebook.com/whitehat>

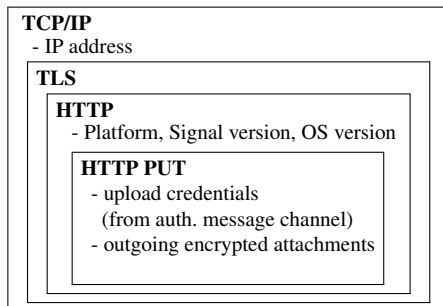
(1) Unauthenticated message channel



(2) Authenticated message channel



(3) Authenticated attachment channel



(4) Unauthenticated attachment channel

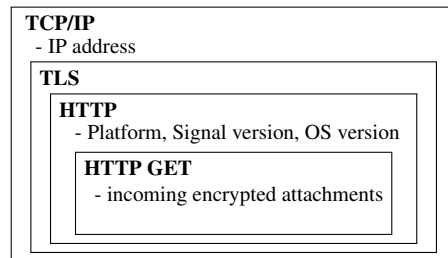


Figure 4.1: The connections and protocols used in Signal, along with notable data provided by the client at each layer. Not all fields appear in all messages (only at some point in the same TCP stream), and many fields immaterial to metadata analysis are elided.

Table 4.1: Entropy at which a user becomes unique (in expectation).

Among all 40 million Signal users [Cur24]	25.3 bits
Among all 2 million Tor users [Tor24]	20.9 bits
10% of the above users	−3.3 bits
1% of the above users	−6.6 bits

Transport Layer Leakage

To fully understand the threat of information leakage from the transport layer, it is important to first understand the sort of protection Tor provides. When proxying a client’s traffic over Tor, the destination server will see TCP streams that appear to originate from one of the available exit relays, selected at random (see Chapter 2). When using applications that tightly integrate with Tor, which exit relay a stream originates from is typically explicitly managed in some way—e.g., in Tor Browser, traffic to different first-party domains will use different circuits, and therefore independently sampled exit relays. If the `IsolateDestAddr` option is enabled for the Tor client, then all streams to distinct IP addresses will similarly use different circuits. This option is not enabled by default, including in Orbot, the most popular general-purpose Tor application for Android, but our analysis will assume users take the strongest defensive posture available (without modifying any source code), and will enable this option. Unless an application is programmed to do otherwise, existing TCP streams will continue to use the same exit relay until they are terminated; new TCP streams will rotate their circuits every 10 minutes. Because of this behavior, and because Signal mostly uses long-running connections that originate at the same time, a Signal client configured to proxy through Tor (e.g., via Orbot) will have the TCP streams for the messaging channels originate from the same IP address, and the two attachment channels from another IP address.

With this in mind, we can estimate the leakage from the transport layer when using Signal over Tor. For group communications, the leakage is total—as all messages leave and arrive on a single authenticated TCP stream, the server can trivially observe all metadata for group communications. To estimate the information leakage in dyadic conversations, we apply an analysis akin to that used in browser fingerprinting [BBB+24; Eck10], and use the Shannon entropy of such a connection, defined as $H(X) = -\sum_x P(X = x) \log_2(P(X = x))$. Simply put, entropy is the expected information content, and the more bits of information available about a connection, the smaller the set of possible users that connection could originate from. For reference, we provide some entropy values at which users become unique in expectation in Table 4.1.

First, we estimate the entropy from the timing of establishing a connection. To know this, we need to know how likely it is that two users establish their message channel connections at the

same time as each other, where “same time” here means within the amount of time it takes for the connections for the two channels to establish from one user. In our experiments, this was always well below 100 ms, as it is effectively the difference in time between the two system threads on the device establishing the connections that determines this, but we will conservatively use 1 second bounds. We do not know how many connections are established per day in practice (they close when network connectivity is lost, then eventually reestablish when the OS messaging API notifies the app of a message or the user opens the app), but we will assume a starting point of one connection per day established at a uniformly random time, giving 16.4 bits of entropy, while each doubling of average connections per day can be modeled by subtracting 1 bit from this value.

For entropy from choice of exit relay, we can use the information available in the network consensus to calculate an accurate value. First, we filter to relays that can be used as exits to Signal, in that they have the flags exit connections require by default (Fast and Exit), and allow connections to Signal’s IP on the standard TLS port [Tor]. Then, we weight them by their consensus bandwidth. Because exit relays are chosen first when building a circuit, this should match the actual probability of the relay being chosen as an exit relay (the network consensus can specify additional weights based on flags, but this is not currently done for viable exit relays). Doing so gives an entropy of 10.7 bits as of this writing. Because the probability is non-uniform, the information content of individual sessions could be higher or lower—e.g., approximately 1 in 280 connections will convey at least 3 additional bits of information by using lower-weighted relays.

Therefore, Signal sessions over Tor have approximately $16.4 + 10.7 = 27.1$ bits of entropy from transport layer leakage in the initial connection—easily enough to uniquely identify users who proxy their connections over Tor, regardless of the fraction who actually do so.

Alternate Sources of Leakage

While the transport layer is sufficient for the server to monitor metadata, other sources of leakage can be observed.

A closely related source to the timing information above, but not shown in Figure 4.1, is the “keepalive” messages the Signal client sends every 30 seconds on both message channels. Investigation of the source code of Signal confirms that the timer used is shared by both channels, and depends only on when the two connections were initialized on the device, so can be used to refine the correlation of the handshake. The two channels are also connected in that if either terminates (e.g., it does not get a response to a keepalive message), the client terminates the other channel and re-initializes them both. This means that the time the connection closed can be used

as a source of entropy, and that an active attacker willing to terminate connections can eliminate possible conflations or confirm suspected associated channels.

Similarly, message traffic patterns can serve as a source of entropy, as each time a user sends or receives a message, delivery receipts in close conjunction provide additional opportunities for correlation. The entropy of these correlations is more difficult to estimate, as unlike TCP connections, users are likely to send and receive many messages per day, and the probabilities are no longer independent, as messages often occur in bursts as a part of conversations. While it would be possible to form some sort of estimate using available data (e.g., some of the sources listed in Section 4.4.2), it would be non-trivial to calculate, and given that our proposed solutions (see Section 4.3.3) to the transport leakage already sufficient for attacks would also apply to this leakage, there is little reason to do so. Suffice it to say, while delivery receipts do see larger delays than simultaneously established connections, especially when using Tor, the empirical results of Martiny et al. [MKA+21] and our own simulated results in Section 4.4.4 show that well over 90% of delivery receipts are received within 2 seconds (from the vantage of the sending client—the server will observe approximately half that delay from the time it forwards the message to receiving the receipt), leading us to believe in the exceedingly rare cases where two users connect from the same exit relay at the same time, even one message would be enough to distinguish between them. Alternatively, even if one million users managed to connect via the same exit relay at the same time, the statistical disclosure attack presented by Martiny et al. [MKA+21] (in this case, binned by TCP stream rather than “mailbox”) requires on the order of 5 messages to uniquely identify participants in a dyadic conversation.

Additionally, there is some metadata provided by the Signal client to the server in-band: Signal version, operating system version, language, and country/region. The precise entropy of each of these values is only knowable by Signal, who, if they do collect such data, are unlikely to share it. However, we can safely assume this entropy is non-negligible—e.g., assuming Signal’s distribution of Android versions among its users is similar to that of the wider Android ecosystem, we would expect about 2.9 bits of entropy from that field as of May 2024 [Sta24a]. While some of this information, such as language, seems to be sent explicitly, other fields, such as the operating system version, appear to be created by the platform’s web request library.

Solutions

The work of Martiny et al. [MKA+21] proposes a solution to their statistical disclosure attacks in the form of “ephemeral mailboxes”, where Signal users create new identifiers to receive messages for each conversation. We agree that such a change would be necessary for Signal to truly protect user metadata against an adversarial server. However, our observations here emphasize that this

design change *would not be sufficient*, and that changes to the implementation details of Signal are necessary as well. As currently implemented, even if ephemeral mailboxes were added, using Tor would merely hide the user’s IP address, and still allow a malicious server to associate all other user metadata to the user’s Signal account, and thus to their phone number.¹⁷

To allow users to opt-in to protecting their metadata, in addition to the aforementioned ephemeral mailboxes, Signal would need to allow configuring Tor as a proxy, and establish independent circuits for each of these mailboxes. Doing so would eliminate IP address as one of the strongest methods of linking connections. In addition, subject to any insight Signal has on the entropy of other sources of identifying information, the fingerprint of each connection should likely be reduced. While the platform and Signal version are presumably useful information for internal purposes, users should be provided a clear way to strip fingerprintable data, such as country/region. As for timing leakage in establishing connections, delivery receipts make it unlikely any low latency and low bandwidth technique will eliminate the ability to link a mailbox’s incoming and outgoing connections. The most obvious solution is therefore ensuring there is no correlation between the timing of the channels for distinct mailboxes. Signal could refrain from establishing these connections until a message is received in the respective conversation,¹⁸ the respective conversation is opened in the app, or on random timers.

4.3.4 Tor-based apps

In the case of apps built around Tor onion services as message channels, there is little metadata leakage to analyze. So long as Tor is secure in the threat model it uses, and so long as that threat model is sufficient for users (e.g., they do not need protection from the global passive adversaries that mixnets protect against but Tor does not), the anonymized peer-to-peer connections make it unclear what party metadata could even be potentially leaked to.

For Briar, the analysis is the same when using onion services, but has a more nuanced level of risk on local networks: without the ability to use onion routing, message metadata is visible as cleartext to any network-level adversary, but such an adversary must have access to the local network. However, in such a scenario, the threat model of Tor would also be violated, as the local network adversary would have passive access to both ends of the Tor circuit, and could perform packet analysis to confirm parties on the local network are communicating [Dan04; DMS04].

¹⁷Signal uses phone numbers as account identifiers internally. Support for usernames was recently added, but these still always map to a phone number while in use on the Signal server.

¹⁸While out-of-band message waking is provided by Google and Apple for Android and iOS respectively, such services may come with their own leakage [Wyd23], and providing such a service in a privacy-preserving and energy-conscious manner is an open problem.

Therefore, when used in local messaging scenarios, Briar is slightly less effective at protecting metadata than other Tor-based messengers if too few messages are sent (relative to any other local Tor traffic) for timing attacks to be effective.

4.4 Performance Analyses of Messengers and Tor

While individual users can, to varying extents, gain metadata protection by relying on the Tor network, the value of such protection is directly connected to the number of users also using that same technique to protect their connections. For example, even if Signal were to adopt design decisions to fully eliminate correlations between channels, if only one user of Signal enables Tor for all their connections, that user has practically just changed their IP address—the server would be able to identify all connections from Tor as being from that user, and still link all metadata (aside from metadata directly leaked by IP address) to that user. Therefore, the usability and performance of Tor and the applications using it are integral to the privacy Tor can provide [ADS03; DM06; FGK+10].

However, Signal’s developers in particular have expressed doubt in the viability of scaling a messenger app on Tor [Mar19]. To judge how much metadata protection is possible, we must therefore have an idea of how well these systems perform, and in particular, how using Tor to proxy them affects the usability of these messengers, and conversely, how the additional messenger load impacts the existing Tor network. Towards that end, we look at the two primary modes of delivering messages via Tor: the more popular central-server messengers (e.g., WhatsApp, Signal), and peer-to-peer onion service messengers (e.g., Ricochet, Cwtch). We therefore ran a series of simulations to see how the network behaves under various configurations using the techniques established in Chapter 3.

4.4.1 Client Model

While Shadow allows us to run actual client code, such as the tor application, we still must rely on user models to create network traffic. These models should be realistic enough that they generate valid results, but efficient enough that we can properly scale them.

To represent the Tor network as it currently exists, both as a control experiment and as background traffic for our messaging experiments, we may rely on the techniques from Chapter 3. To study the properties of adding messaging traffic in particular, we must create a similarly appropriate model. However, unlike the models presented so far for simulating Tor traffic, which are derived from observational studies on real Tor traffic, we are now attempting to model a form

Table 4.2: Distributions and probabilities in the model.

Dist.	Description
I	Time <i>Idle</i> before sending a message.
I'	I with each value’s weight scaled by that value.
W	Time <i>Active</i> without activity before becoming <i>Idle</i> .
A_S	Time <i>Active</i> to send a message after sending one.
A_R	Time <i>Active</i> to send a message after receiving one.
S	Pr. of <i>Idle</i> to <i>Active</i> transition after a message is sent.
R	Pr. of <i>Idle</i> to <i>Active</i> transition after a message is received.

of traffic which does not yet exist on Tor in significant quantities. Therefore, while TGen can simply represent abstract network flows observed on exit relays, we do not have access to such data to train models for this new traffic. We must therefore instead construct a user model from available data on application use, generating traffic from the “bottom up” instead of a “top down” view. To do so, we designed and implemented a simulated messaging client and server, which makes use of Markov models to generate realistic messenger traffic as simply as possible—namely, we rely on sampling existing data while adding just enough machinery to avoid degenerate edge cases (e.g., a sampled user who only sent two successive messages should not constantly generate traffic). Future work may improve on creating more accurate conversational traffic.

We begin by assuming we will only be simulating at most one hour of traffic, and are not generating traffic for all users, only concurrently connected users in such a timespan. Conceptually, each client in the simulation is representing a user who is actively monitoring their device for messages over the course of the hour. However, because the source of data for our Markov models (see Section 4.4.2) does not select for users active in a particular hour, we still sample from all active users—i.e., all users in the dataset, which is roughly equivalent to the standard industry notion of Monthly Active Users (MAU). When modeling our users, because behavior outside the scope of an hour is outside the scope of the simulation, we are allowed performance shortcuts, such as removing all conversations that would take over an hour to begin from the simulation entirely.

We perform two styles of experiment: client-server, and peer-to-peer.

For the client-server model, no onion services are simulated. There are two centralized servers in these experiments: a custom server with a binary protocol for quickly relaying messages, including delivery receipts, and a simple HTTP server for relaying message attachments. The servers have bandwidths set arbitrarily high so as to never bottleneck. While the servers could, in reality, be the bottleneck, given that the current systems are deployed and functioning without issue, it is almost certainly the case that the messaging service can handle the required traffic, and

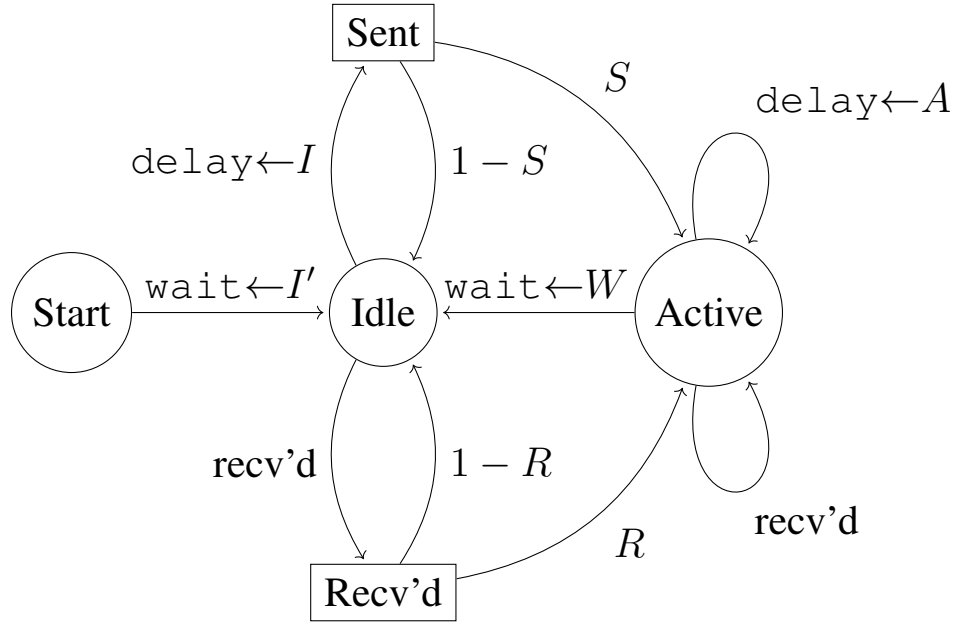


Figure 4.2: State diagram for a user’s conversation model. Expressions involving S and R are probabilities, `delay` and `wait` are sampled durations, `recv’d` means a message was received.

this is not what we are testing.

Clients register the list of groups they are in with the server during the bootstrap phase. These registrations are not intended to accurately reflect dynamic group membership, and we assume that the changes in real-world group membership during the simulated hour would be negligible for overall performance.

For the peer-to-peer model, clients are behind an onion service, and messages are sent directly to onion addresses.

In both models, a set of groups is constructed, with membership and sizes sampled from some distribution, including dyadic groups of size 2 (see Section 4.4.2 for the distribution used in this work). Clients are configured with these groups, and associate with each group they are in four distributions (I , W , A_R , A_S) and two probability constants (S and R), all summarized in Table 4.2. In the centralized model, each client connects to the server with one Tor circuit per conversation, and registers each connection with an identifier, which can then be routed to with a lookup table on the server (in a real server, these identifiers would be the opaque ephemeral mailboxes described in the work by Martiny et al. [MKA+21], but for our purposes, human-readable identifiers for easier debugging and analysis are fine). If any connection is closed, the client immediately reconnects. In the peer-to-peer model, clients use onion addresses as their contact identifiers.

The Markov model consists of three primary states (*Start*, *Idle*, and *Active*) and two secondary states (*Sent* and *Received*), where “primary” states are the states clients spend most of their time in, and “secondary” states are intermediary states used to set values and determine state transitions. To help visualize the state transitions described below, a visualization is provided in Figure 4.2. Once all clients have completed the above bootstrap phase as part of the *Start* state, each client conversation (the group state from a client’s perspective) samples an initial `wait` value uniformly from the interval $(0, d)$, where d is sampled from I' , a distribution of idle times weighted by that idle time—e.g., if I (used later) was a 50/50 distribution of one minute or one hour, then for I' , sampling one hour would be sixty times as likely as one minute. This initial wait is the only value we sample weighted by duration, since the probability of the experiment starting in any particular sampled idle time is proportional not only to the number of times that idle time occurred, but how long that idle time is. After taking no actions in this initial `wait`, the conversation enters the *Idle* state.

When transitioning a conversation to the *Idle* state from *Start*, *Sent*, or *Active*, a new `delay` value is set via sampling from its I distribution, representing the amount of time in seconds until the user sends a message on this conversation with no other impetus. Upon sending a message in the *Idle* state, the conversation transitions to the *Sent* state. It then transitions to the *Active* state with probability S (representing the user intending to imminently continue sending messages in this conversation); else, the conversation re-transitions back to the *Idle* state (the user only intends to send the one message for now). Upon receiving a message in the *Idle* state, the conversation transitions to the *Received* state. It then transitions to the *Active* state with probability R (the user intends to respond to this message), else it transitions back to the *Idle* state (the user does not yet intend to respond to the message), though it continues to use the previous value of `delay` in the latter case.

When transitioning from *Sent* to *Active* or from *Received* to *Active*, a `wait` value is set via sampling from W , representing the maximum amount of time in seconds the user will wait after any activity in this conversation before it transitions to *Idle*. When transitioning to the *Active* state from sending a message, whether from *Sent* or *Active*, `delay` is set via sampling from A_S . When transitioning to the *Active* state from receiving a message, whether from *Received* or *Active*, `delay` is set via sampling from A_R . After `delay` seconds, the client sends a message, and transitions to the *Active* state. Whenever an *Active* conversation receives a message, it again transitions to the *Active* state.

4.4.2 Data Sources

To perform the experiments, we first have to obtain data to instantiate the distributions and probabilities described above. In addition, we need a source of data about group sizes, and the sizes of the messages that are sent.

For the distributions and probabilities listed in Table 4.2, we can make use of the “Share and Multiply” dataset made publicly available by Anika Seufert et al. [SPSH23]. The “Share and Multiply” dataset consists of 5,956 WhatsApp chat logs, donated by users and anonymized by the researchers. It contains ordered transcripts of the donated conversations, with message contents replaced with data such as message length, and sender identifiers replaced with an unspecified hash function. (No data about recipients is given, so we assume that all participants sent at least one message.) The dataset includes information on 76,720,159 messages from 117,695 WhatsApp users.

We wrote a series of scripts to process the dataset and produce files representing the distributions and probabilities of Table 4.2 in a form usable by our simulated chat client (described in the next section). Rather than attempt to determine analytic forms of each distribution in question, we instead opted to have each simulated user sample from the indexed empirical distributions of a randomly selected user from the dataset. This is, rather than attempt to find a probability distribution that fits the distribution of idle times well, we simply store all the times a user was idle between messages, and sample from that distribution during execution. Because we are only selecting the distributions representing user behavior, and not the behavior itself or the set of chats a user is in, this means that even simulated users who sample the same user from the dataset will (with very high probability) behave differently. While our client implementations do support a wide variety of analytic distributions, opting for this design removes much of the guesswork while still providing the necessary variability in behavior.

However, to use the above technique, we still need some way of distinguishing messages sent while “active” vs. “idle”. To do this, for each user, we take all sent messages, bin them according to time sent, fit a two-state Hidden Markov Model with Poisson emissions to these bins, label the longer-tailed state as the Idle state, and label all sent messages accordingly. We then filter down to users who sent at least two messages while Idle, two messages while Active, or replied to a message while Active (users who only sent messages while in one state are configured with a distribution where they never enter the other state).

We use data from research by Michael Seufert et al. [SSHT15]—which presents statistics from comprehensive WhatsApp app data (rather than individual conversations) donated by 209 participants—to determine the number of dyadic and group conversations a simulated user is in, and the “Share and Multiply” dataset to determine the number of participants (up to 255) in the

non-dyadic groups. This gives us a set of unpopulated conversations for each user, each of which we combine with other users’ conversations that have the same target number of participants, until each group reaches this target. After, each target size’s remaining group that has insufficient participants (if any) is padded out with randomly selected simulated users who receive that group’s messages but never send their own.

For message sizes, we need information on text message lengths, on file attachment sizes, and file attachment frequency. The “Share and Multiply” dataset contains information on message lengths and attachment frequency (but not on attachment sizes). However, in the process of analyzing the dataset, we observed that 317,758 messages (0.4%) listed their lengths as negative. Because message lengths are given with the number of emoji subtracted, and because all messages where this negative length occurs had a positive emoji count, we believe this is the result of mixing two mechanisms of measuring string lengths (e.g., code points and graphemes). The authors of the dataset were unable to confirm or deny this as the cause, but regardless, because message lengths on the network are binned by padding length (160 bytes for Signal), emoji are unlikely to comprise the majority of long messages, and most messages were less than one padding length regardless, we believe adding the message length and emoji count then rounding up to the next positive padding block length is sufficient for our purposes.

For file attachment sizes, we use a modified version of the “WhatsApp, Doc?” tool first presented by Garimella and Tyson [GT18]. We used the tool to query the first 256 public group links provided with the source code, indexed in 2018. Of those groups, 85 still existed as public groups, which we joined using the tool. We stayed in these public groups for approximately two months, then extracted the file sizes of all 3,112 attachments sent, creating a single index distribution, discarding all other data. For each simulated user, when creating the indexed distribution of message sizes, if the corresponding message from the “Share and Multiply” dataset had a file attachment, we add a file of a size sampled from this distribution of file sizes.

4.4.3 Implementation

We implemented our simulated messaging clients and servers, which we call **mgen**,¹⁹ for use in Shadow. There are four executables that comprise mgen: **mgen-client**, which simulates clients in client-server experiments, **mgen-server**, which forwards messages to mgen-clients, **mgen-web**, which provides an HTTP server for mgen-clients to upload and download message attachments, and **mgen-peer**, which handles all traffic and behavior in peer-to-peer simulations.

Because of the constraints of large-scale simulations, the implementation was created with performance and memory overhead in mind. All components are written in Rust, to take advantage

¹⁹<https://git-crysp.uwaterloo.ca/j3tracey/MGen>

of its low-cost abstractions and library ecosystem. We also make use of a technique established in Section 3.3.2, where traffic from multiple users can be simulated from a single mgen and tor client. Specifically, because each conversation is over a separate Tor circuit, there is effectively no difference between a set of clients on the same Autonomous System, and an unusually connected and high-volume client (except that the former would have multiple circuits to the same recipient, which our implementation also allows). Because of these implementation details, we are able to scale experiments far beyond what would be capable with naive or off-the-shelf messenger implementations. While this technique was effective, limitations in the performance of the Tor client required us to share circuits among multiple connections for one of our experiment configurations (see Section 4.4.4).

The client-server implementation (and, to a lesser extent, the peer-to-peer implementation) is designed to accurately model the network performance characteristics of Signal. Message paddings are configured to match those used by Signal, the threshold for message lengths that force file attachment behavior are designed to approximate Signal's, and all connections are tunneled over TLS. The clients for the client-server simulations are written to allow for any number of each type of server, but because the servers are already multi-threaded with minimal scaling overhead, with uncapped simulated bandwidth, we only scale this count to avoid port exhaustion. Future work could make use of our implementation to perform experiments involving network timings with multiple servers, e.g., comparing Signal and WhatsApp's topologies [SKBP23].

There are some notable differences between our behavior and Signal's. To cater to the most privacy-preserving configuration and to simplify the Markov models, we do not simulate read receipts or typing indicators (delivery receipts are simulated, with the simplified model of all active users having internet connectivity during the simulated hour). Additionally, we do not simulate voice or video calls, which make up approximately 61% of Signal's bandwidth costs [WL23]. We do not simulate this traffic because Signal's calling functionality, like most modern internet calling services, relies on WebRTC, which in turn uses UDP rather than TCP streams for the bulk of the traffic. (Strictly speaking, the specification for WebRTC does allow TCP as transport, but this is rarely supported in practice.) Since Tor cannot currently be used for relaying UDP traffic, it is not possible to use Tor to tunnel Signal calls, and there is no reason to include calls in the simulation. Finally, we only intend mgen to simulate network behavior—outside the TLS and Tor links, no cryptography is used, and like all Shadow experiments, computational performance impact is considered out of scope.

Table 4.3: Shadow experiment configurations.

Model	Proxy	Circuits	Users [†]
client-server	none	-	10,000
client-server	none	-	100,000
client-server	Tor	indep.	10,000
client-server	Tor	shared	10,000
client-server	Tor	shared	100,000
onion service	Tor	shared	10,000

[†] All networks are sampled at 10% scale; real-world equivalent is $10\times$ given value.

4.4.4 Results

With the resources available to us, we were able to run 10 sampled 10% scale Tor networks for each of 6 configurations, listed in Table 4.3. These networks were sampled according to the principles established in Section 3.3. They were generated, parsed, and plotted using *tornettools* (slightly modified to insert *mgen* processes and uncap server node bandwidth), along with our own tools for *mgen* statistics. All charts are presented with 95% confidence intervals (lines that appear to be missing said intervals are too tightly distributed to see).

There were two limiting constraints on scale. The first was the tor processes, which were not written to handle the large number of circuits from simulating many users on each client, and encountered severe computational performance degradation. When using Shadow in the more typical Tor net configuration, relying solely on TGen for generating traffic, additional load is simulated by increasing the traffic roughly linearly with the number of circuits—a number typically far fewer than needed for the large circuit counts used by theoretical users of a Tor-integrated client-server secure chat app. To attempt to circumvent this constraint, we patched the Tor client by replacing constant-time operations with more typical libc short-circuiting functions. We do not need the additional security granted by using these constant-time operations, and computation time is not simulated in Shadow anyway,²⁰ so this was a pure performance improvement. We also increased the number of Tor processes on each client host, so that each was handling traffic for fewer circuits. This revealed the second limiting constraint, which is the number of system threads that can run a single Linux system. Regardless of configuration, Linux on a 64-bit system will not permit more than $2^{22} \approx 4$ million system threads [Lin23]. Because each simulated process is an actual process since Shadow 2.0 [JNW22], this limits the scale of the experiment we can run, particularly if we wish to run multiple experiments in parallel on one machine.

To scale even larger, we therefore ran two experiments that shared circuits instead of using

²⁰This is also why the degraded computational performance of Tor does not affect accuracy.

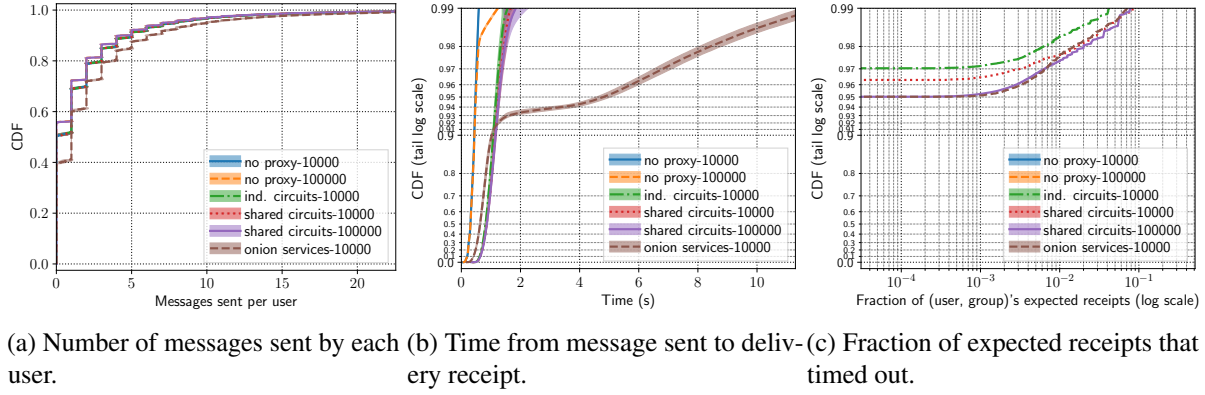


Figure 4.3: The impact of Tor on messenger behavior. The count of messages sent is unaffected by Tor in Fig. 4.3a, the delay and pattern of delivery receipt timing varies depending on configuration in Fig. 4.3b, and Tor impacts timeout rates (shown on the horizontal axis) with high variance in the tail of Fig. 4.3c.

unique ones for each client: one experiment with a smaller user count to compare it with an experiment without this change, and one with ten times as many users. Because the results of the smaller user count with and without this technique were always well within each other’s error bars, this grants some confidence that the larger experiment similarly reflects the results without this change. Because there is no server to hide metadata from in the onion service model, connections are already shared on a per-user basis (i.e., messages from one user to another will use the same circuit, regardless of whether the messages are for separate groups).

The listed user counts are users in the simulation; just as for TGen user counts, because they are in a 10% scale network, they are simulating the equivalent of 10 times that many users in the real world. Therefore, simulating 100,000 users in a 10% scale network from Table 4.3 will yield results corresponding to one million real world users, or about 1 in 40 Signal users according to the latest available estimates [Cur24].

4.4.5 Effects of Tor on Messengers

First, we examine the effects on the users of the simulated messenger application. In Figure 4.3a, we see nearly all configurations exhibit effectively identical distributions of messages sent over the course of the examined time period. This is because while we have thousands of distributions of data to sample from, we are sampling 10,000 or 100,000 times, pushing towards the central limit of client behavior. The onion service model exhibited a very small but statistically significant increase in messages sent due to a difference in experiment bootstrapping. In the client-server

model, clients must space out registration with the server to avoid filling the (non-configurable) TCP receive buffer, and thus need a non-negligible bootstrap period before messages successfully begin sending and receiving. In the onion service model, there is no central bottleneck for bootstrapping, and connections are generally established successfully whenever the respective user models cause a message to be sent or received. Even though all presented data only comes from after 20 simulated minutes into all experiments (much longer than needed to fully bootstrap), clients begin in the *Idle* state, and the earlier conversations for onion service users leads to a slightly larger number of *Active* clients at the start of the measured 40 simulated minutes.

In Figure 4.3b, we see the time it takes for a user to receive a delivery receipt for a message they sent, from the time the original message was sent, excluding messages whose delivery receipts timed out (defined as taking longer than 30 seconds). When using Tor as a simple proxy, a statistically significant amount of delay is added, but generally less than one second's worth, taking less than two seconds total. Onion service clients, on the other hand, produce a distinctive bimodal distribution. For over 90% of messages, the time from sending a message to receiving a delivery receipt is even less than the time it takes when using Tor as a simple proxy. It is likely this is largely a result of the number of network hops involved in the respective connections—when both clients are using Tor, a proxied connection has 9 nodes (2 clients, 6 relays, 1 server), requiring 8 links, while when using onion services, there is no server, eliminating one node and one link from the connection (see Figure 2.1 and Figure 2.2). The remaining receipts, however, can take notably longer, following a much longer-tailed distribution, with over 1% of messages taking over 10 seconds to receive a receipt. This is a result of the peer-to-peer nature of these onion service clients: while a client-server model will typically make use of long-lived connections to the central server, as the server would be unable to deliver messages without one, onion service clients only establish a connection to a peer upon sending or receiving a message. Thus, if the message sent is to a recipient that has not recently had messages sent to or from the sending client, the time it takes to establish the connection will be added to the time it takes to receive a delivery receipt.

However, the additional delay is not the entire effect on user experience. Figure 4.3c shows the fraction of receipts that timed out of the receipts expected by a user in a conversation—e.g., if a user sent two messages in a group with ten other users, and exactly one delivery receipt took longer than 30 seconds to arrive for one of those messages, the fraction would be 0.05 for that user in that conversation. While the large variance between networks prevents us from making any rigorous claims comparing the rates between Tor configurations, with the non-proxied experiments consistently showing 0 timeouts, we can say that using Tor increased the timeout rate of receipts, potentially up to 10% for 1% of users. As noted in Figure 4.3a, the overwhelming majority of users send fewer than 10 messages in the course of the 40 simulated minutes, likely accounting for some of the observed variance, and possibly inflating the fraction among the users who sent fewer

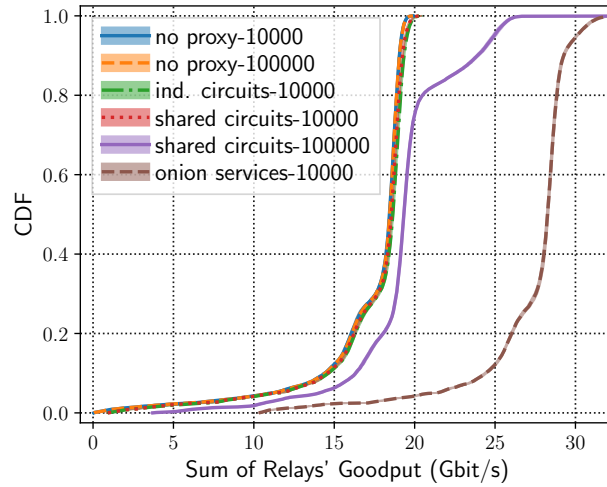


Figure 4.4: The cumulative goodput of all relays. Onion service messenger users in particular dramatically increase network load.

messages and saw a timeout (e.g., if timeouts are low probability but uniformly distributed among messages, then experiments where timeouts happen to be more concentrated in low-message users will appear as more prone to users with lots of timeouts than experiments where timeouts happened to concentrate in users that send many messages). Regardless, the effect is consistent enough that it is unlikely to be a statistical aberration.

4.4.6 Effects of Messengers on Tor

Next, we examine the effects of additional load on Tor. In Figure 4.4, we show the cumulative goodput of all relays in the network. While proxying the equivalent of 100,000 users' client-server message traffic had a negligible effect on total Tor traffic, scaling to the equivalent of 1,000,000 users did show a statistically significant increase in load, and the equivalent of 100,000 users of onion service messengers dramatically increased the goodput at all percentiles. Such a large increase in traffic from the peer-to-peer nature of onion service messengers is not entirely surprising, given that the client must send a message to each member of a group chat individually, rather than letting the server duplicate the traffic for the respective recipients—a cost exacerbated when sending larger messages with file attachments. Despite this additional load, Figure 4.5 shows no significant impact on download times for 0.5 and 1 MiB files by TGen (i.e., non-messenger) clients. This makes some sense, as much of the additional load will be over middle relays, which are less constrained than exit relays (exit relays being less popular to operate, as they are identified

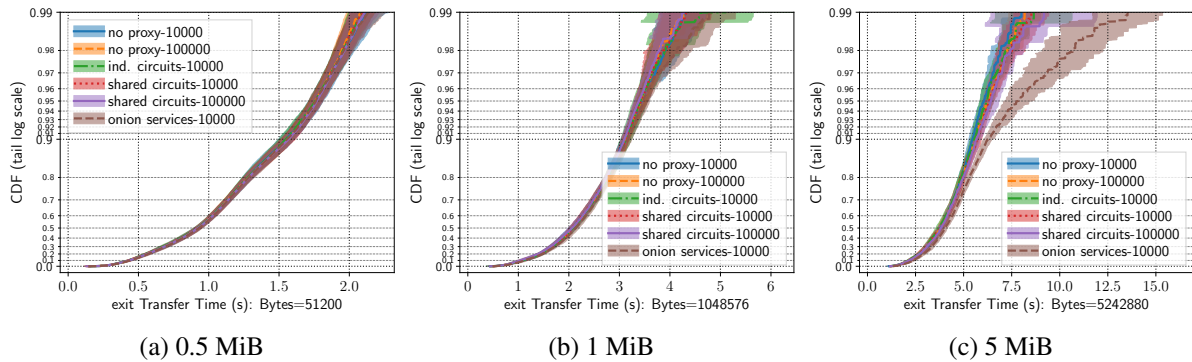


Figure 4.5: The impact of Tor-enabled messengers on non-messenger Tor performance. Time to last byte in seconds of 0.5, 1, and 5 MiB downloads from measurement TGen clients. Smaller downloads are unaffected by the greater load, but the slowest 5 MiB downloads see noticeable slowdown with additional background onion service messenger traffic.

as the origin of Tor traffic by naive classification methods). However, the additional load from onion service traffic did become notable for 5 MiB downloads, with the slowest 1% of downloads taking an additional 4 seconds (50%) longer on average. All other Tor performance metrics, such as circuit build times and error rates, were unaffected by the additional load.

4.4.7 Discussion

The implications these results have on Tor as a means of metadata protection in messengers is open to some interpretation. The delays in the client-server model to delivery receipts could theoretically prove unacceptable for some users, but we believe the additional second is unlikely to be especially notable for most users. The potential for 10 seconds of delay when using onion services is obviously far more noticeable for users, but it is worth reiterating that these delays are generally a result of initiating new conversations. This could make such delays more tolerable, as even 10 seconds of round-trip delay, while large from a network perspective, may be socially less important when parties are not in the midst of an active conversation. Unlike small to moderate delays, a particular user failing to receive confirmation their message was delivered 10% of the time could easily be seen as unacceptable. The number of users who seem to be affected by these timeouts appears proportionally small, but with large networks, 1% of users is still notable. Similarly, that the results could be skewed by the frequently small number of messages sent, the point remains that a user who, e.g., only sent two messages and failed to receive delivery confirmation on one could have a non-negligible chance of preferring to switch to not using Tor (whether through disabling it in some setting, or using another messenger entirely).

The impact on the existing Tor network is similarly somewhat ambiguous. While such costs on the tail end of the traffic distribution could be considered acceptable, it seems likely that this would have at least some noticeable impact by current users of Tor. For example, larger web pages could take longer to load, and streaming video content would be far more likely to require buffering over the course of the video. Such changes can decrease the number of Tor users [AAH13; FGK+10], reducing the privacy of the remaining users [ADS03]. Therefore, while there is no sign that Tor would fundamentally break at these deployment scales, we nevertheless would recommend that any integration of Tor into a messenger with a large userbase be orchestrated with a corresponding increase in relay capacity.

4.5 Conclusion

Widespread adoption of metadata protection remains an open problem in the space of secure messengers. While Tor is often presented as a mechanism users can already take to protect themselves, we have shown that the reality is not so simple. Taking existing clients and relaying their traffic over the Tor network does not eliminate the metadata created and sent (explicitly or implicitly) on those connections, and onion services currently seem unlikely to scale to the size of a platform like Signal without adversely affecting the current uses of Tor. Addressing the former will require considerable care in engineering these messenger applications, while addressing the latter will require further research in Tor performance or a sizable increase in the capacity of the Tor network. Until then, the common suggestion to simply run messenger services with Tor will not accomplish what is likely intended by such advice, and if used widely, could be detrimental to other users of Tor.

Many questions remain unanswered in this subject. We have yet to scale to the entirety of Signal’s userbase relying on Tor, much less WhatsApp, or Facebook’s new e2ee messaging deployment. The impact of arti—Tor’s new implementation in the Rust programming language, still in the process of gaining support for onion services—on both simulation scale and network performance remains to be seen. Nor have we yet been able to test the impact of more complicated onion service messenger architectures that have been proposed, such as the previously mentioned Cwtch group servers, or Quiet’s use of IPFS.

In an age of frequent security breaches, surveillance by authoritarian regimes, and monetization of sensitive personal information, protection of metadata around communication is more important than ever before. Just as the original deployment of usable end-to-end encryption in popular messaging applications took concerted effort in both engineering and research, now is the time to design and develop usable techniques for investigating and achieving adoption of metadata protection in our online communications. For now, we note that while the design of metadata

protection in centralized systems relies on defenses that could be accidentally negated with the simple change of an HTTP stack, integrating defenses via onion services into the core of the design makes such mistakes exceedingly unlikely.

Chapter 5

The Impact of Rust on First-Time Code Contributors

We have demonstrated lowered barriers to entry for PETs research on Tor in Chapter 3 (by establishing sound experimentation practices), and shown in Chapter 4 how Tor can be used to improve safety and lower barriers to entry for PETs research and development (by showing the circumstances in which Tor is a viable means of simplifying the research and development of metadata protection in e2ee messengers). The remaining question we answer, in this chapter, is how development practices can improve safety and lower barriers to entry for Tor development. This question is of great importance, as one of the simplest ways to ensure that the users of a project can continue to use and engage with the project as they see fit is to maintain low barriers to entry for making these direct contributions to the project. When a small number of contributors wield the majority of development contributions, the direction a project can take can suffer from a lack of diversity. Project enhancements are less likely to be implemented when “heroes” dominate development in open source projects [ARK+18], and developers for projects, including the Tor Project, have struggled to secure funding for maintenance work that is less attractive to traditional funding sources [Smi19]. While not all users will have the ability or desire to write code for a project, the more we allow those who are in a position to contribute their time and skills to readily do so, the more likely it is we can help keep contributions diverse, and maintain a healthy developer community. Here, we examine the ability of first-time contributors to make contributions to open source software, with an eye on PETs in particular.

While allowing for new contributors to make their first contributions is beneficial, it can also be difficult, even prohibitively so, for project maintainers to achieve this goal. Work by Asparouhova (*née* Eghbal) observes that project maintainers are often reluctant to engage in encouragement of new contributors:

...in speaking to maintainers privately, I learned that these [new contributor] initiatives frequently cause them to seize with anxiety, because such initiatives often attract low-quality contributions. This creates more work for maintainers—all contributions, after all, must be reviewed before they are accepted. Maintainers frequently lack infrastructure to bring these contributors into a “contributor community”... [Egh20, Introduction]

Asparouhova further notes that security in particular can be a challenge, since “... security vulnerabilities can be time-consuming to manage, with little upside for the developer, coupled with the fear of an extremely bad situation if they miss something important” [Egh20, 04: Work Required by Software]. In practice, this manifests as most projects having a small number of contributors contributing most commits, with one study finding that one-time contributors, despite making up nearly half of all contributors in a selection of open source projects, contributed less than 2% of commits [PSG16], another finding 77% of examined open source projects followed an 80-20 rule (i.e., at least 80% of contributions were made by at most 20% of contributors) [ARK+18], and many other similar results [RM10; SWCG13].

In the case of PETs like Tor (or the software they build on, such as Tor Browser’s use of Firefox), concern over the security of contributions is even more present. The tools provided are used in highly adversarial settings, where untrusted inputs are commonly received remotely, and many attackers have a vested, well-funded interest in compromise. As such, patches from new contributors can truly require the greater scrutiny common to them. However, there are also problems with relying on a small number of contributors for this scrutiny, particularly in combination with concerns about homogeneous funding models, which can bias shepherding contributions towards the problems most relevant to that source of funding, rather than what is most needed by the community (see, e.g., the aforementioned funding difficulties at the Tor Project).

Therefore, such projects would benefit from new techniques for lowering the barrier to entry for contributing code to their software. One potential avenue for lowering this barrier to entry is taking software written in memory-unsafe languages (such as the core Tor router, Firefox, and the Tor Browser software derived from them), and porting components or their entirety to Rust. Rust is a newer systems programming language, with greater memory safety guarantees than C or C++,¹ as well as general language design decisions that can reduce the incidence of vulnerabilities of software written in it [MK14].² While there have been concerns about the use of Rust discouraging new contributors to security-critical projects [FCV+21], the Tor Project has recently claimed that the safety properties of Rust have helped make new volunteer contributions

¹For the remainder of this thesis, we will use C++ to concisely refer to C as well.

²<https://www.rust-lang.org/>

more common [Mat22]. In this chapter, we examine the question of whether new contributors are less likely to contribute vulnerabilities when using Rust than C++, and the impact that has on new contributors.

To answer this question, we use the results of the Oxidation project [Moz20], which seeks to replace components of the Mozilla Firefox web browser written in C++ with equivalent components written in Rust. By using such components, we can ensure that the comparisons made between the two implementations are as fair as possible for real-world projects, as they are designed to be in-place replacements for their respective role. The comparisons we make between the projects are performed by estimating learning curves from extracted data about vulnerabilities, and the commits that introduced them. In the process, we improved the tools to extract such data, and created a dataset that can be used and further contributed to by future research.

We start by reviewing some related work in Section 5.1, which will provide some of the necessary background into the terminology and techniques we use. Then, in Section 5.2, we describe the data sources, extraction techniques, processing, and analysis methodology. We present the results of our analysis in Section 5.3, and conclude in Section 5.4. The scripts and hand-annotated data necessary to reproduce our results are publicly available.³

5.1 Background and Related Work

5.1.1 Identifying Vulnerability Fixes

A well-known technique for identifying buggy changes is known as SZZ [ŠZZ05]. While the original paper does not focus on vulnerabilities, but instead bug-introducing changes, it is commonly used as a technique for correlating bugs with the commits that introduced them (in our case, commits known as Vulnerability Contributing Commits, or VCCs). In essence, the SZZ algorithm consists of identifying relevant bugs, identifying the fix commits associated with those bugs, then using the appropriate `annotate` command in the revision control system (e.g., `git's annotate/blame`) to identify the most recent commits that modified the lines either modified by or adjacent to the lines of the fix commit, and assigning those commits as inducers. That is, for each removed line, and lines near a line added or removed in the fix commit, find the most recent commit that changed that particular line. From this, we obtain a list of commits that nominally contributed to the bug, which are called “fix-inducing” commits (in our case, they are also the VCCs), and the information associated with those commits (authors, times, histories, etc.).

³<https://git-crysp.uwaterloo.ca/j3tracey/grading-on-a-curve>

While SZZ is a common approach for identifying commits that introduced bugs, it is not without its shortcomings. For one, the technique casts a wide net on blame—depending on its configuration, it includes not only who wrote the lines that were changed, but also any lines near it, as well as non-semantic changes, such as variable renaming. Therefore, it is likely that much of the data includes authors who were not at all to blame for the introduction of the original vulnerability, and simply modified code adjacent to where the fix was applied, or refactored the code and preserved the bug [DMS+16; KZPW06].

Another major problem with this approach is that it is not always the case that fixes are applied in the same place where a bug was introduced [DMS+16]. Suppose, for example, as part of a fix for the particular vulnerability, code needed to be introduced that generated a compiler warning. As part of the fixing commit, another part of the file needed to be changed such that compiler warnings did not occur. While changing this part of the file was ultimately necessary, none of the code around this part of the file was ultimately to blame for the original vulnerability—it was the lack of code added that created the problem, not the code in that part of the file. Therefore, the above approach would erroneously mark the author of this portion of the file as a contributor to the bug.

Various modifications to SZZ over the years have been made to attempt to address some of these and similar shortcomings. For example, if a fix commit also happens to add a comment to some non-buggy code, a naive implementation may attribute the adjacent unmodified lines, so better implementations are syntax-aware [WS08]. Despite these changes, SZZ still has its limits, but in any case, it remains one of, if not the most popular technique for identifying bug-inducing commits.

Another approach, used specifically for identifying VCCs and not general bugs, was used in the work of Meneely et al. [MSM+13]. In this paper, the approach used was to examine the vulnerability being fixed, write a script that would detect that specific vulnerability, then bisect the revision control system with said script to identify which commit introduced it (commands such as `git bisect` will run the specified script in a binary search pattern on revisions depending on the result). This approach, while highly accurate in a certain sense, requires understanding each vulnerability in the data set being analyzed. In the case of the cited work, it took three researchers “hundreds of man-hours over six months” to analyze 83 vulnerabilities. Furthermore, it has a very narrow definition of what introduced a vulnerability. Because the vulnerability has to be exploitable, the buggy code could be hidden behind a feature flag while some feature is under development, or be unexploitable until executed in a particular way, and the bisection will detect the commit that exposed the vulnerability rather than introduced it in the sense relevant to code review. Ultimately, this technique was determined to be a poor fit for our purposes, though we do use manual identification of vulnerabilities (see Section 5.2.1 and Section 5.2.2).

The research of Perl et al. [PDS+15] aimed to provide a database of vulnerabilities in open source projects, including Firefox, but it appears to be no longer available. In any case, their technique appears to closely match that of SZZ. In the paper, they examined, among other questions, whether “new committers are more likely to introduce security bugs than frequent contributors”. They found that “new contributors” are five times as likely to introduce vulnerabilities, though they define “new contributor” by the fraction of commits to the project overall—not, as one might expect, the number of commits at the time the vulnerability was introduced.

5.1.2 Correlating Vulnerabilities with Contributor Experience

While there are many papers that examine vulnerabilities, e.g., predicting or identifying parts of code liable to have them [NZHZ07; WSS14], most do not address the matter of their relationship with the experience of the developer at the time of introducing said vulnerability. Of those that do, most found that there is some negative correlation between experience and introducing vulnerabilities [BCH+14; MW00; PDS+15], though at least one study found no strong correlation with bugs in general [RD11], while another study found a small but positive correlation with vulnerabilities in the use of Java cryptographic libraries [HGK+19, III.B.1]. Rather than being contradictory, we believe these results stem from the nature of vulnerabilities, and their intersection with memory safety—for details, see our analysis in Section 5.3.3.

5.1.3 Oxidation

Early in Rust’s history, it was adopted as a research project by Mozilla as a means of looking into techniques for increasing safety in systems programming languages [Hoa10]. The initial flagship use of the language was an experimental web browser, named Servo.⁴ As the language matured, Mozilla decided to replace components of Firefox with equivalents written in Rust (as well as write new components in Rust), a project named Oxidation. As of this writing, 25 components are listed on the Oxidation page as having been shipped [Moz20].

This undertaking provides a useful source of data, since it means we can compare the C++ and Rust versions of each re-implemented component, and how they are developed, directly. However, not all of these components are completely germane to a study of vulnerabilities in systems languages. As mentioned, some components are entirely new, and therefore have no point of comparison. Similarly, some components replaced original versions written in a memory-safe language generally not considered a systems language (JavaScript), and were replaced with Rust

⁴<https://servo.org/>

Table 5.1: Oxidation projects for which we were able to make comparisons between C++ originals and Rust replacements.

Component	Original	Replacement	Replaced
MP4 parser	stagefright	mp4parse-rust	2016
Unicode encoder	uconv	encoding_rs	2017
CSS styling	style	Stylo	2017
Rendering	Layers	WebRender	2019
Encoding detection	chardet	chardetng	2019
Hyphenation	libhyphen	mapped_hyph	2020
MacOS audio	cubeb_audiounit	cubeb-coreaudio-rs	2020
Color management	qcms	qcms	2020

for reasons other than memory safety. Finally, for our comparison, we require that there be at least one identified vulnerability in either the original C++ or Rust equivalent.

Ultimately, we identified the components in Table 5.1 as valid for comparison. Of those projects, all C++ versions are tracked within the main Gecko repository (the core of Mozilla’s projects, including Firefox), while all Rust equivalents are tracked as their own projects, with the exception of portions of WebRender that are used to bind to the rest of the Gecko code, and Stylo, which was originally a component of Servo, and is now primarily maintained in a fork of Servo tracked in the main Gecko repository. Two C++ components, stagefright and libhyphen, were vendored from other projects (Android and Hunspell, respectively), then tracked within the main Gecko repository, albeit with patches frequently adapted from upstream.

As shown in Table 5.1, Oxidation began shipping in release versions of Firefox in 2016, and the compared components were replaced in 2020 or earlier.⁵ Since all measured commits were authored since 2012 (see Section 5.2.1), this means that the projects are studied over roughly comparable development time frames (4–8 years for C++, and 3–7 years for Rust).

- **MP4 parser:** Parsing MP4 (i.e., common video) files.
- **Unicode encoder:** Encoding of Unicode strings.
- **CSS styling:** Applying CSS rules. The Rust version, Stylo, was taken directly from Servo.

⁵Components compared are every component valid for comparison up through 2023; the only additional C++ component ported to Rust since then (the crash reporter) was launched in April 2024, so has not had enough time for any vulnerabilities to be discovered.

- **Rendering:** The core rendering engine for the browser. The Rust version, like Stylo, was taken directly from the Servo browser. Unlike Stylo, it is now tracked directly within Mozilla’s Gecko repo.
- **Encoding detection:** Detecting which text encoding a page is using.
- **Hyphenation:** Choosing where in wrapped text to insert hyphens.
- **MacOS audio:** Audio support for MacOS.
- **Color management:** Management of color profiles.

5.2 Methodology

Because of the differences in how the components are tracked, our methodology is split into a description of how we extracted data from the original C++ code, and how we extracted data from the Rust equivalents.

5.2.1 Data Extraction: Original Code

To analyze the data available from the original C++ projects, we use two broad steps: first, we identify vulnerabilities in the respective projects in Bugzilla, the issue tracking system used for Mozilla projects; second, we manually identify which commits, from the perspective of a code reviewer, introduced the vulnerabilities in question.

Identifying vulnerabilities

Bugzilla⁶ is the primary means of tracking issues and submitting changes to the Firefox code base. With few exceptions, before any commit can be added to the Firefox release branches, it must have an associated bug on Bugzilla, identified with an integer ID (though one bug may have several associated commits). These bugs are classified into “products”, such as “firefox” for bugs only relevant to Firefox itself, “thunderbird” for Mozilla’s mail client, and “core” for bugs in components shared across products. Bugs are then further classified into “components”, which approximately correspond to the scale of the components that were replaced in the Oxidation project (e.g., “CSS Parsing”). Each bug then has some additional metadata, including the “type”

⁶Specifically, Mozilla’s instance of the Bugzilla product. <https://bugzilla.mozilla.org/>

(e.g., “defect” or “enhancement”), the branches affected, the priority, the severity, people assigned to the bug, and “keywords” such as “crash” or “regression”. Bugs can be commented on, and given attachments, usually consisting of patches, reproduction scripts, or logs. Eventually, one of these patch attachments is accepted by a reviewer, and it is added as a commit to one or more branches. In the course of a normal bug, at this point, a “status” field in the bug is changed to “resolved”, and a “resolution” field changed to “FIXED”.

Since 2012, Mozilla has tracked security bugs by adding particular keywords to them. For our purposes, all of the relevant security bugs can be found by searching for one of the severity impact levels: sec-critical, sec-high, sec-moderate, or sec-low. Prior to 2012, security bugs were tracked by being in a “Security” component. This means we do not have component information for security bugs from that time, so we opt to exclude any bugs that were introduced before this date. (One may think we could instead filter out bugs that were *identified* prior to this date, but this would bias our results by not being able to distinguish VCCs from that time vs. non-VCCs. If we were to include only the commits that were marked as VCCs from before 2012, this would exclude the large number of non-VCCs from that time, biasing the proportion of vulnerabilities per commit higher. If we were to include all the other commits from before 2012, assuming they were not VCCs, this would mean that vulnerabilities Mozilla identified from that time would be left out, artificially lowering the proportion of vulnerabilities per commit.)

To identify relevant vulnerabilities then, we use the Bugzilla API to fetch all bugs with a security level keyword in the component that most closely corresponds to the component replaced as part of Oxidation. Once the IDs of the bugs are found, we identify the comments on that bug that changed the resolution to “FIXED”, extract the linked patches from them, and find the patches’ commit messages in our local repository (we do this rather than using the commit identifiers directly since while Firefox uses the Mercurial revision control system for development, our SZZ implementation is designed to work with Git, which the Rust project replacements use, as does Mozilla’s official Firefox mirror on GitHub). This provides us with a set of fix commits. In some cases, this is sufficient, but in cases where Oxidation replaced only a subset of the component, we then perform additional filtering by the files that were replaced, removing any bugs whose fix commits did not touch the relevant files.

Identifying VCCs

Our initial attempt at VCC identification made use of SZZ. We modified the open source SZZ Unleashed implementation [BSBH19] to function with C++-syntax aware diffs, allowing it to ignore comments, whitespace, and preprocessor changes, and also modified it to ignore any changes to non-C++ files. We also made various changes to the behavior that we found were

necessary to produce better results; e.g., the upstream version only attributes commits from lines removed/changed, but since vulnerabilities are often caused by missing code, such as bounds checks, we also attribute the commits that introduced the lines immediately above and below any new lines in the fix commit.

Some investigation into the SZZ results indicated that the precision of the technique was too low to fully trust. Another possible source of information on vulnerability introducers is Mozilla’s Security Approval forms. In December of 2012, Mozilla added a requirement that whenever a patch fixing a sec-critical or sec-high bug was to be merged into a release branch, rather than a development branch, a Security Approval form must be filled out in the comments of the bug to demonstrate that it would be done responsibly (so as to avoid publishing patches that attentive attackers could use to identify the vulnerability, and exploit it on still-vulnerable branches—such Bugzilla bugs are kept private until some time has passed). When a vulnerability only affects some supported branches, one field of the form requires identifying the Bugzilla ID of the bug whose associated patch made the vulnerability exploitable. This allows reliably identifying which branches require backporting the fix patch.

However, these forms do not align with what we are trying to ascertain either, since for our purposes, the notion of a VCC should be from the perspective of a code reviewer—the VCC should be the commit where human interpretation of the modified code would consider it to be a security flaw, not by the code that enabled the vulnerability to be exploited in production builds of Firefox. For example, many vulnerabilities were hidden behind feature flags while new functionality was under development, and therefore could not execute in normal Firefox builds. In our analysis, these vulnerabilities would be introduced when some C++ code was committed that, when executed, could cause an exploit to succeed—not the commit that flipped the feature flag that allowed the vulnerable code to execute. Similar to the Meneely et al. method [MSM+13] described in Section 5.1.1, which relied on program execution to identify VCCs, Mozilla’s Security Approval forms are concerned with exploitability, not with the code review process that led to the vulnerability.

Instead, we opted for a manual review of the issues in question; i.e., we read the relevant information on the issue tracker, understood what the vulnerability consisted of, then went through the history of the files until we found a commit that introduced the vulnerability from the perspective of a code reviewer. It is our belief that fully automating this process would be extremely difficult. In one case, we identified the VCC as a commit adding a code comment claiming that a particular structure was safely serializable, when it contained fields that made this untrue. Such a change, in a vacuum, has no effect on the behavior of the program—a well-written SZZ implementation would ignore the comment, the Meneely et al. method would never identify such a change as a VCC, and the Mozilla Security Approval forms would never bother to identify that particular change as relevant for backporting. For the purposes of someone reviewing commits

to the relevant code, however, one would expect the reviewer of the commit adding the incorrect comment to identify that serialization is not safe, not the reviewer of the commit later making use of this documented claim that made such a mistake actually exploitable.

Since we do have a working SZZ implementation tailored to this case, and ground truth in the form of our manual review, we have an opportunity to measure the efficacy of SZZ. The comparison of the results of these two methods can be found in Section 5.3.1.

5.2.2 Data Extraction: Rust Equivalents

Unlike the original C++ components, most of the Rust replacements created as part of Oxidation are not developed in the same repository as Firefox, and they typically do not use Bugzilla for tracking issues or contributions. Instead, each project is maintained with varying degrees of independence in individual GitHub repositories. Because of this relative independence, vulnerability issues are not tracked with the same procedures across projects, let alone with the same methods as the original projects. As such, our procedures for identifying vulnerabilities was ad-hoc, relying on whatever we found to be most effective for each project. In practice, the most useful resource was still Bugzilla, as bugs that manifested as exploitable vulnerabilities in Firefox were tracked there, but with the resolution being to use a sufficiently recent version of the project that had addressed it, not fixing the code that contained the vulnerability. Once we were able to identify such a tracked bug, however, we could then investigate where the relevant bug was tracked in the respective GitHub issue tracker for that project.

Our methodology for identifying VCCs in the Rust projects from that point on was similar to the procedure for identifying VCCs in C++. Once the issues were identified, we used information from the issue and its pull request (on GitHub) to manually identify which commit introduced the vulnerability. We provide additional analysis of these Rust vulnerabilities and their causes in Section 5.3.1.

5.2.3 Examining Experience

For our purposes, the matter of most importance is the safety of contributions from first-time contributors. We therefore need some mechanism of quantifying the relationship between experience and safety, so that it may be compared across languages. To achieve this, we make use of established research into learning curves [AF11], though used here for the novel purpose of comparing properties of programming languages. Learning curves are used to study the relationship between experience—typically measured in some form of repetitions or exposure to some

task—and some other value—typically an amount of time, or a probability of success/failure. Here, we are studying the relationship between developer experience with a particular project, and the probability of introducing a vulnerability. In this work, we measure experience as the number of commits the contributor made, at the time of the contribution, to the project. (As such, “experience” here is an analogue for familiarity with a codebase and development practices for a project—not experience with the language overall. This is a metric one would expect to correlate with code quality, and makes particular sense for our purposes, as not only is project data more accessible for researchers, it is also what project maintainers would know when performing code review of a merge request.) If the learning curve for a project has a more negative slope than the learning curve of another project, it indicates that contributors to this project more quickly learn to avoid adding vulnerabilities than the other project.⁷ More importantly for our purposes, if the y -intercept of a project’s learning curve is lower than that of another project, it indicates that first-time contributors to this project are less likely to introduce vulnerabilities.

Of course, real world learning is more complicated than simple, single-variable functional relationships; e.g., people learn in different ways, forgetting and time between tasks can be included into the model, and actual data is generally noisy. Despite the shortcomings, learning curves have been found to perform well in extremely diverse fields, including in measuring performance of programming tasks [CA94]. In this research, we opted to rely on what has historically been the most popular learning curve model [AF11; NR81], a power law of the form

$$P_j = P_1 j^{-\ell}$$

where P_j is, in our context, the probability of a vulnerability in a commit from a contributor’s j th commit, and ℓ is a “learning rate” constant. P_1 is the subject of our interest—the probability a contributor’s first commit contains a vulnerability.

We should emphasize here that despite being a power law that tells us probabilities, this is not a power law *distribution*. The curve is not itself a probability distribution, and it will not sum to 1; rather, it is a function that takes as inputs experience values, and produces Bernoulli distributions.

Traditionally, the empirically estimated terms in the power law learning curve (P_1 and ℓ) are found by taking the log of the empirical data, giving data in the presumed form

$$\log P_j = \log(P_1) + \ell \log x$$

and performing a linear regression. Unfortunately for our purposes, but fortunately for the security of Firefox, there are not enough vulnerabilities at every experience level to perform such a regression (nearly all experience levels have 0 vulnerabilities, and nearly all of those that do will

⁷Note that this, like most actual learning curves, is inverted from the colloquial sense of a “steep learning curve”.

have 1). To accommodate this, we instead look at cumulative data, which allows us to express it with larger, more continuous values. However, we must express the accumulated data in a form that accounts for the fact that our empirical data is a result of a particular distribution of experiences in the respective repository. To incorporate this fact, we define

$$V_{\leq j} = \sum_{k=1}^j P_k c_k$$

where c_k is the number of commits made with exactly k experience in the project (again, defined as the number of commits the author has made to the project), and $V_{\leq j}$ is the expected number of vulnerabilities from contributors with at most j experience. We also define $v_{\leq j}$ as the empirical value for $V_{\leq j}$ (i.e., the actual number of vulnerabilities at or below j experience).

To estimate the P_1 and ℓ parameters, we perform a gradient descent search. We define our loss function for each experience j as:

$$L_j(P_1, \ell) = (V_{\leq j} - v_{\leq j})^2 = \left(P_1 \sum_{k=1}^j c_k k^{-\ell} - v_{\leq j} \right)^2$$

(with the total loss being the sum over all j).

We compute the gradient $\nabla L_j(P_1, \ell)$ as:

$$\begin{aligned} \frac{\partial}{\partial P_1} L_j(P_1, \ell) &= 2(V_{\leq j} - v_{\leq j}) \sum_{k=1}^j c_k k^{-\ell} \\ \frac{\partial}{\partial \ell} L_j(P_1, \ell) &= 2(V_{\leq j} - v_{\leq j}) P_1 \sum_{k=1}^j -c_k k^{-\ell} \log k \end{aligned}$$

(where again, the gradient of the cumulative loss function is summed over all j).

In practice, the search frequently found early local minima, so an array of plausible starting values were used.

Counting Experience

For the C++ projects, experience is counted over all of Firefox, from its inception, while samples (i.e., the commits counted as either containing or not containing vulnerabilities) are scoped to commits made since 2012 to the files in the examined project (see Section 5.2.1). We believe this

corresponds with how project maintainers experience contributor experience; e.g., a contributor making their first contribution to a particular component is likely to be more trusted if they are a regular contributor to other parts of Firefox than if they have never contributed at all. However, if this not true, the bias would be towards greater experience, which would lower the C++ P_1 value—i.e., if this assumption is false, then we can expect the real value of the C++ P_1 to be even higher than our results. A similar effect applies to the skewing of Firefox contributors towards higher experience as a result of excluding samples but including commits from before 2012 as experience.

For the Rust components, which are mostly developed as independent projects, experience is counted only from the respective repository. For Stylo, we use all commits to the Servo project (in a repository broken out from the Gecko fork, and in the case of determining samples, again scoped to the relevant files—see Section 5.1.3 for additional context). The binding portions of WebRender tracked in Gecko are handled in the same way as C++ projects. For all other Rust projects, the git repository and project map one-to-one. It is worth noting that the respective Rust projects were typically initiated by Mozilla developers familiar with the domain space, so experience in some sense will be undercounted (albeit with a different language). Such undercounting should be minimal, as the small set of maintainers will quickly cease to be counted as new contributors—e.g., every maintainer can only make their first commit once, and new contributors will be a much larger set than the set of maintainers.

For both C++ and Rust projects, merge commits are excluded as samples.

5.3 Results

5.3.1 Identified Vulnerabilities

A summary of the vulnerability counts themselves can be found in Table 5.2, which includes the number of samples (i.e., the number of commits in the project), the number of tracked issues identified as potentially fitting our criteria, and the number of commits that could successfully be confirmed to contain at least one C++ or Rust vulnerability (VCCs). In the case of the C++ data, the gap between the number of vulnerability issues and the number of VCCs is primarily from bugs where the attributed commits were introduced prior to when security vulnerabilities began being tracked, as described in Section 5.2.1. The number of VCCs is greater than the number of issues for the MP4 parser (i.e., stagefright) code because of an issue fixing multiple vulnerabilities of a similar form that were introduced in two independent commits. For the Rust data, the gap (on a per-project basis) is mostly a result of vulnerabilities that were introduced in C++ code used to interface

Table 5.2: Sample size (commits in project), vulnerability issues, and valid vulnerable commits for C++ and Rust projects. Empty cells are invalid to count against the total, e.g., because there were no VCCs in either the C++ or Rust versions of the projects; see Section 5.3.1.

	C++ Projects			Rust Projects		
	samples (commits)	Bugzilla security issues	positives (VCCs)	samples (commits)	Bugzilla security issues	positives (VCCs)
MP4 parser	776	16	17	910	0	0
Unicode encoder	337	5	1	879	0	0
CSS styling	3152	11	9	5764	8	5
Rendering	6177	21	19	7011	7	7
Encoding detect.	—	2	0	—	0	0
Hyphenation	164	2	2	50	0	0
MacOS audio	134	1	1	841	2	2
Color mgmt.	—	7	0	—	1	0
Crossbeam	—	—	—	—	2	—
Total	10740	65	49	15455	20	14

with Firefox. We also note the Rust project Crossbeam at the bottom, which is not comparable, as it is a third-party dependency (specifically, it is a library with a collection of concurrency tools). It is included here because the noted vulnerabilities were tracked downstream by Mozilla, and because Firefox’s C++ code typically vendors dependencies (as mentioned in Section 5.1.3, this is how stagefright and libhyphen were initially incorporated as Firefox components). Therefore, even though we cannot compare Crossbeam with any corresponding project, its vulnerabilities are nevertheless noted, since, if there were equivalent vulnerabilities in C++, they likely would have been in Firefox’s code, and possibly tracked in one of the equivalent components we do compare against. Put another way, not including such vulnerabilities would allow one to nominally eliminate *all* Rust vulnerabilities by moving all code, other than library imports, to third-party dependencies.

Any component where the original and Rust project both had no attributed vulnerabilities were left out of the analysis—if there were vulnerabilities we could not include, there was no way to create a fair comparison, and if there were no vulnerabilities at all, then it indicated it was not a project where security was prone to being a problem.

Rust Vulnerabilities

To provide more insight into the nature of Rust vulnerabilities, we here detail some observed properties of the 20 identified vulnerabilities. All vulnerabilities are identified by their Bugzilla IDs. For a description of each bug individually, we also provide a full enumeration in Appendix B.

First, there is a notable distinction for what qualified as a vulnerability in a C++ or Rust project. In Rust, there exists a notion of “soundness”, which is a guarantee that all code, excluding lines explicitly annotated as “unsafe”, cannot have undefined behavior [Uns23]. That is, soundness implies that if some Rust code does not make use of the `unsafe` keyword, and it compiles, then the code is well defined. Soundness bugs, then, are bugs that allow this guarantee to be violated in some way—e.g., a bug that allows some safe code written in a particular way to cause a null pointer to dereference somewhere. In the Rust ecosystem, soundness bugs are largely treated as security vulnerabilities, as they can violate the memory safety of the language, which can in turn be used to exploit traditional memory safety vulnerabilities. C++, being memory-unsafe, does not have an equivalent notion—it is akin to marking nearly *all* code as unsafe, since any API (functions, methods, constructors, etc.) that makes use of a pointer to access memory can trivially be made undefined by supplying it an invalid pointer. It is therefore worth noting that many Rust vulnerabilities would likely not be considered vulnerabilities at all had they been written in C++, since the vulnerability tracked is merely stating that it is, e.g., possible to use a library in such a way that it would allow memory corruption, and not that the library had actually been used in such a way that memory corruption could occur in execution. Regardless, we treat such vulnerabilities the same as any other, since the subject under study is not whether code is exploitable in practice, but the impact the threat of vulnerabilities has on accepting code from new contributors—a project maintainer is tasked with managing reports of vulnerable code, not exploits themselves. In our data set, three vulnerabilities were explicitly tracked as soundness bugs, including both Crossbeam vulnerabilities (1668514, 1716028), and one in WebRender (1685145).

In three cases, a vulnerability was introduced that had existed in the C++ equivalent being replaced. Two of these vulnerabilities were found in the Rust code, and only then was the same or a very similar vulnerability found to have existed in the C++ code as well. In the case of 1614971, this fix was backported, and so is tracked as both a C++ and Rust vulnerability in our data, but in the case of 1622291, the fix was never backported, and so is only counted as a Rust vulnerability in our data. The remaining vulnerability (1420001) was fixed in the C++ code, and never existed in the Rust code’s original context in Stylo, but was reintroduced in Firefox as a result of Firefox’s configuration. Because most C++ vulnerabilities were memory-safety vulnerabilities (at least 87% in our data), and Rust is a memory-safe language, it is expected that most C++ vulnerabilities would not translate into identical Rust vulnerabilities.

Six vulnerabilities involved some form of direct interaction with C++ code (1557208, 1577439,

1593865, 1614971, 1622291, 1758223), functionality known as Foreign Function Interfacing (FFI). In every case, this was caused by a race condition with C++ threads. These races typically occurred during browser shutdown, while freeing all resources, and four of these vulnerabilities were determined by Mozilla developers to be unlikely to be exploitable in practice, but were tracked as security vulnerabilities to err on the side of caution.

Seven vulnerabilities were in the parsing of web page content or its rendering. Three of these (1599181, 1602843, 1680084) were in the parsing and sanitizing of CSS copied and pasted by the user. An additional three involved the rendering of the page escaping certain sandboxing assumptions, such as the ability to render invalid graphics data (1637112) or to render page contents outside the page boundaries (1700235, 1701834). The seventh (1631232) involved an error in the implementation of a garbage collector for CSS rules.

Only two bugs do not fit into at least one of the above descriptions. An example of a bug that could occur regardless of language choice, the VCC introducing 1746545 modified a build script (written in Rust) to add a `-ffast-math` compiler flag to a compiler invocation, which then had the unforeseen effect of the compiler optimizing out checks in the source code for run-time floating point errors. Conversely, the other bug, 1696312, occurred as a result of unsafe code added for higher performance memory caching—the sort of unsafe code invocation future compiler optimizations or language features could hopefully obviate.

SZZ

For our comparison with SZZ, we did not constrain ourselves to the commits that were valid for comparison with Rust, but instead included all VCCs we identified (e.g., VCCs from before 2012 are included). SZZ attributed a total of 130 commits over 43 issues, with a maximum of 14 commits attributed to one issue, versus 77 commits over 54 issues from our manual review, with a maximum of 4 commits attributed to one issue. 31 of these commits were attributed by both SZZ and our manual review. In 11 issues, SZZ attributed the same set of commits our manual review identified. In 5 issues, SZZ attributed a strict subset of the commits our manual review identified. In 12 issues, SZZ attributed a strict superset of the commits our manual review identified. In 2 more issues, SZZ attributed some other non-zero intersection of commits our manual review identified. Together, this means SZZ has a non-zero intersection with our manual review in 30 issues, or 70% of the issues it was able to attribute a non-zero number of commits. This makes SZZ significantly better than random guessing, but especially with the 26% rate of total matching per-issue, 24% rate of matching per-commit, and lower rate of overall resolution of issues to attributed commits, the validity of SZZ for any research continues to require justification on a case-by-case basis, as previously established in the literature [DMS+16; KZPW06; WS08].

While we do not make any specific quantitative claims about the causes of the discrepancies between the two, common causes seemed to match known problems with SZZ: improper attribution to cosmetic or other unrelated changes, and bugs induced by *not* adding code to some other location in the repository, so that the fix was applied in an area untouched by the inducer.

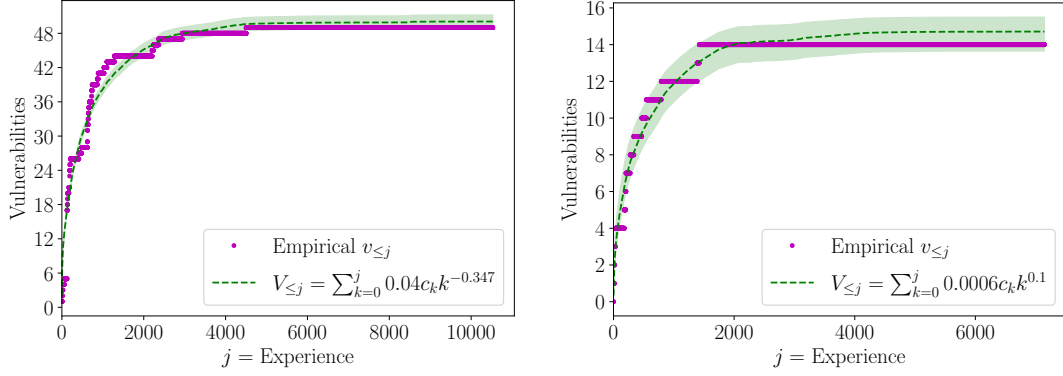
5.3.2 Learning Parameters

The results of our gradient descent can be seen in Figure 5.1, where the empirical relationship between project commits and vulnerability count is plotted against what is predicted by our model, while Figure 5.2 shows the respective learning curves. (Note that Figure 5.1a and Figure 5.1b are not directly comparable, as the number of commits, c_k , differ—e.g., a project with many first time contributors would bias the expected vulnerabilities at $j = 1$ higher, regardless of the true value of P_1 .) The upper and lower error bounds in all three figures correspond to the bounded sensitivity of the model—i.e., the maximum difference any single false positive or negative could shift the best fit. In practice, this corresponds to counting a non-VCC with experience 1 as a VCC or counting the lowest experience VCC as a non-VCC, for the upper and lower bounds, respectively. These error bounds appear tight, especially in Figure 5.2, as should be expected—even though there are relatively few vulnerabilities in absolute number, the relevant files collectively have tens of thousands of commits to them, each of which is included as data in our models. That is, the best fit parameters are over a distribution with low (but far from negligible) probability over most of its realistic domain, but with an extremely high number of samples, ultimately resulting in a good estimate of the desired parameters. Or, put simply, if the real values were far larger or smaller than our estimates, we would expect to see far greater or fewer vulnerabilities over those tens of thousands of commits, with high probability.

With the given parameters, the gradient descent procedure found that the best fit parameters for the C++ data gives an intercept $P_1 = 0.04 \pm 0.01$ and a learning rate of $\ell = 0.347 \pm 0.001$, for a learning curve of $P = 0.04j^{-0.347}$, while with the Rust data, $P_1 = 0.0006 \pm 0.0003$ and $\ell = -0.1 \pm 0.1$, for a learning curve of $P = 0.0006j^{0.1}$.

5.3.3 Analysis

First-time Rust contributions were significantly safer than their C++ equivalents The y -intercepts of the two learning curves imply that for Oxidation projects, a first-time contributor to a C++ project was approximately 70 times as likely to introduce a vulnerability as a first-time contributor to an equivalent Rust project. This provides strong evidence that even if one were to



(a) C++ contributor experience vs. vulnerabilities. (b) Rust contributor experience vs. vulnerabilities.

Figure 5.1: Contributor experience vs. the number of vulnerabilities introduced. Each dot is the number of vulnerabilities introduced by the set of commits that are some author's j th or lower commit to the project.

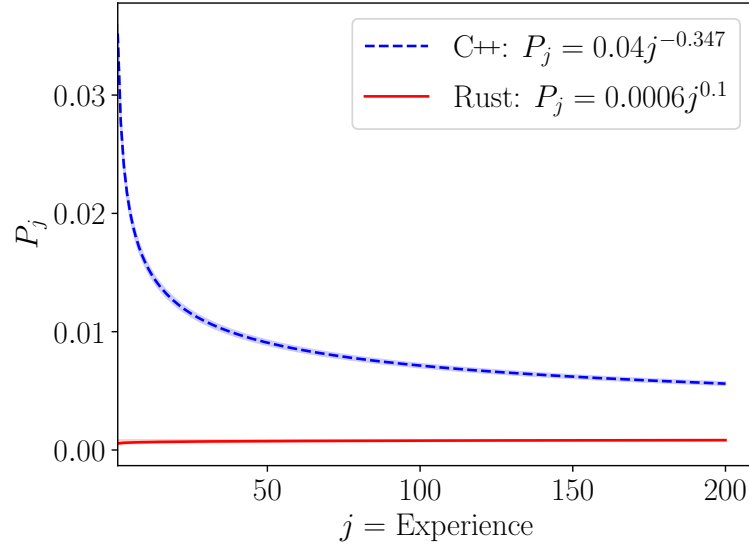


Figure 5.2: The C++ and Rust learning curves. P_j is the probability a commit from an author with j experience will introduce a vulnerability.

accept that Rust is a more difficult language to learn than C++, it can still provide a sizable net benefit to new contributors to such projects.

Memory safety may change the fundamental relationship between experience and vulnerabilities Aside from the implications of the respective P_1 values, the difference in learning rates is also of note. While the C++ projects have a typical (albeit slow) learning rate, the learning rate of the Rust projects is negative (i.e., experience is raised to a positive power). The phenomenon is robust—flipping the sign of the best-fit ℓ from negative to positive (though a still very small value) would require adding at least three 1-experience VCCs to the Rust data set; i.e., a 21% overall increase in vulnerabilities, all concentrated at the absolute lowest experience value (a very unlikely phenomenon). A negative learning rate is highly unusual for most contexts, as it means the probability of an error is *positively* correlated with experience. For our data, this means that while C++ developers become less likely to contribute a vulnerability in any particular commit the more commits they have already made, a Rust developer is more likely to contribute a vulnerability as they gain more experience with a project. This counter-intuitive phenomenon has some precedence: as mentioned in Section 5.1.2, a study on the use of cryptographic libraries in Java found a similar small positive correlation [HGK+19, III.B.1], while other studies found the expected negative correlation with other C++ projects [BCH+14; MW00]. From this, an explanation presents itself: since the largest class of vulnerabilities are memory safety bugs, which can be overlooked even in simple code, these vulnerabilities mask the more general phenomenon of vulnerabilities that are more likely to present themselves in innately challenging parts of the code. For example, while at least 87% of the C++ vulnerabilities we identified were bugs derived from memory unsafety (and even more were only vulnerabilities because of memory unsafety; e.g., integer overflows), the majority of vulnerabilities found in the Rust projects were in interfacing with C++ code or the implementation of a language parser—both notoriously difficult problems (see Section 5.3.1). Because of this difficulty, new contributors are less likely to make changes to these parts of the code, making what vulnerabilities that do occur positively correlated with expertise and experience. Similarly, usage of cryptographic libraries in Java (a memory-safe language) would also be more prone to vulnerabilities in more difficult parts of the code that are likely the domain of experienced developers. While this notion of complexity is subjective, we expect there would be little contention to the claim that newer contributors prefer and are generally encouraged to focus on starting with simpler contributions.

Future work may yield better results That the learning rate of the C++ projects is larger than the learning rate of the Rust projects means these lines intersect, meaning the models project a quantity of experience (around 18,000 commits) for which a C++ developer will be less likely to introduce a vulnerability than an equivalently experienced Rust developer. That the learning

rates are of different signs means the models project a level of experience where a Rust developer will be more likely to introduce a vulnerability than *any* C++ developer, though the value is astronomically large (around 10^{18} commits). In practice, we believe that such crossovers are unlikely, and instead point to limitations of the power law model. While we could try many other models of learning curve, concerns of overfitting and multiple comparisons would need to be addressed, so we leave as future work analysis that would start with causal explanations for why a particular model would be a better representation of the underlying phenomenon. Until such research occurs, it is of course worth keeping in mind that very few developers create 18,000 commits in any project over its lifetime (the only contributor with more than 18,000 commits in any of the repositories examined was a bot), and no project will ever see 10^{18} commits.

It is worth noting that the empirical data will undercount the number of vulnerabilities in the low-experience portion of the learning curve, precisely because of the phenomenon we are trying to address: contributions from new contributors tend to receive greater scrutiny and more thorough inspection than those from contributors with recognized experience. The goal of this extra scrutiny is, at least in theory, to attempt to level the learning curve to horizontal, and to make all contributions ultimately as safe as feasible. As one might expect, and shown in the above results, such a goal is not always achieved, but it is worth keeping in mind how this weighs on the data. Fortunately, this extra scrutiny is generally only applied to the very low experience portion of the learning curve; most of the curve will have the a typical amount of scrutiny, and since all portions of the curve are given equal weight when performing the gradient descent, we expect this bias to have minimal effect on our results. Namely, while this phenomenon will cause an unfortunate observable discrepancy between empirical low-experience vulnerability proportions and what our model predicts, when this is precisely the experience levels we are most concerned with, the goal is to extrapolate what those values would be *without this extra scrutiny*, making such a divergence expected.

Finally, there is some limitation to these results in that they all come from Oxidation projects. One-to-one replacements of C++ projects in Rust are rare, and it is fortunate we have such a high-quality point of comparison. Ideally, we would also compare widely used or scrutinized C++ projects that were based on original Rust projects, but this is unlikely to happen in practice, as there is little to no advantage to doing such a rewrite, let alone deploying it. Should such projects come to exist, though, they could prove useful to validate these results.

5.3.4 The Prevalence of New Contributors

Given the above results, natural questions to ask are whether the reduced prevalence of vulnerabilities in contributions from new contributors increased the rate of new contributors in practice,

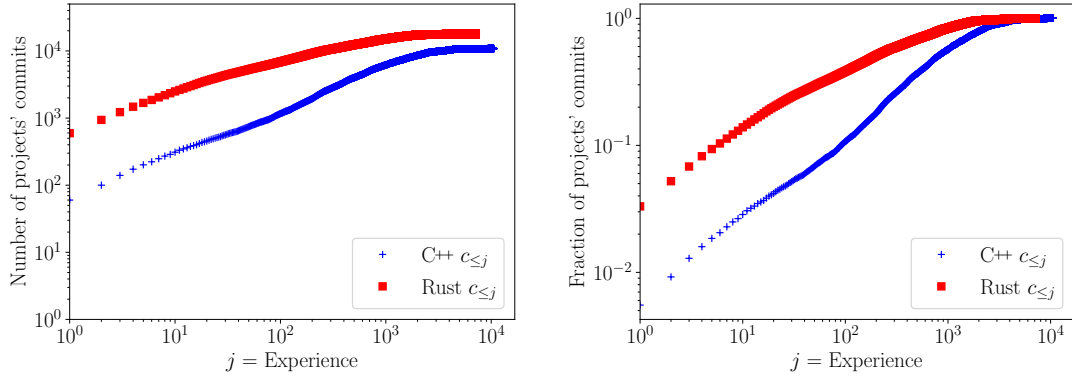


Figure 5.3: The experience of the examined C++ and Rust Oxidation projects. Each mark is at the number (left) or fraction (right) of commits to the projects that are some author’s j th or lower commit to the project. Note the log-log scale used to make lower-experience contributors more visible.

or whether Rust simply acted as its own filter and reduced the rate of new contributors entirely. The former is difficult to say, given the number of factors aside from security that can make Rust attractive to potential developers. However, we can confirm the latter is not the case. In Figure 5.3, we compare the distribution of experience to commits, c_k in Section 5.2.3, in absolute and proportional terms—i.e., the number of commits made to C++ and Rust Oxidation projects (in the examined time frame) by contributors with at most j experience, and the fraction of commits made by contributors with at most j experience. The figures clearly show, in both absolute and relative terms, the Rust projects have more low-experience contributors than the C++ projects they replaced. We can therefore safely say that Rust has not been preventing new contributors, and it is unlikely that the observed effects on safety of contributions from new contributors are the result of a smaller pool of select developers. It is possible Rust developers are more experienced with programming in general, but this is immaterial for the problem being addressed—the goal is not to accept contributions from individuals new to programming, but to allow new contributors to the project to more easily contribute.

5.4 Conclusion

In this chapter, we examined the effect of choice of programming language on the likelihood of new contributors introducing vulnerabilities, using a novel application of learning curves. While previous research has mentioned concerns over the difficulty of learning Rust perhaps presenting

a barrier towards its adoption (specifically when used as a means of increasing the security of applications) [FCV+21], our work shows that the reality is complicated, and gives evidence to the claims made by the Tor Project that Rust can help new contributions [Mat22]. Namely, while it may still be true that Rust may feel like a more difficult language to learn, in at least some ways, new contributors actually benefit from its adoption, with their first contributions being less than 2% as likely to introduce vulnerabilities as C++, and first-time contributors appearing at a notably higher rate in the projects examined. Such safety could potentially lead to less work for maintainers, and empower a wider and more diverse body of contributors to submit changes that are important to them.

Chapter 6

Conclusion

In this thesis, we have examined a diverse set of topics relating to PETs research and development, centering the practices that influence Tor, and the practices that Tor influences. In Chapter 3, we focused on experimentation, and conducting statistically sound research on Tor. In Chapter 4, we looked at the viability of relying on Tor for metadata protection in end-to-end encrypted messengers, in the process making use of the techniques we developed in Chapter 3. In Chapter 5, we investigated developer experience to better understand how switching from a memory unsafe language to Rust—as Tor is in the process of doing—impacts the viability of accepting first-time contributors into a project.

Through this research, we have found strong evidence for the thesis statement:

By improving Tor experimentation methodology, we not only benefited our other research, we also established techniques used as the current *de facto* standard for Tor network simulations [Aro24; DDB23; DSC+22; DT23; JW23; Mic21; OAD22; PAKE23; RE22; WHH22] and that have seen use in studying the performance of censorship circumvention tools [Boc23]. Shadow was already in use as a mechanism of making Tor experimentation safer by avoiding running experiments on the live Tor network (which is actively relied on by users), but by examining how these kinds of experiments impact Tor, we published techniques that make it easier for researchers to approach Tor network experiments in a way that the results are trustworthy as presenting real phenomena, and not statistical coincidence—which, in turn, may hopefully allow for Tor to more readily incorporate lessons from such research into its design.

In looking at the interactions between end-to-end encrypted messengers and Tor, we found that using Tor as a proxy, despite commonly being recommended by e2ee messenger developers [Esr24; Mar19; MKA+21; Mol23], does not eliminate metadata protection concerns when researching or developing such applications. On the other hand, using Tor’s onion service functionality can

abstract away metadata concerns and help developers simplify the research and development of e2ee messengers, subject to the performance implications demonstrated in our experiments. E2ee messenger researchers and developers may therefore use this research when deciding where to dedicate their focus when designing or developing apps, instead of needing to develop expertise in the nuances of metadata protection.

Our investigation of Firefox’s investments in Rust indicate that Tor’s anecdotal observation of a wider body of contributors as a result of porting their C code to Rust is likely to generalize. Something about the differences between C and C++ versus Rust—likely the safety properties the language provides—do seem to make it easier for new contributors to get involved in development, despite concerns raised over the language presenting barriers to adoption by such developers [FCV+21]. Therefore, by using Tor’s development practices for framing our investigation, we were able to discover a technique that could be useful to other PETs developed in C or C++ to address concerns over safety and high barriers to entry for development contributions.

Of course, the work presented here is not the end of the story. This avenue of research is understudied, and countless possible observations, techniques, or motivating projects remain: While our investigation through identifying and studying elements of PETs research and development with noticeable friction did yield worthwhile and actionable results, it is also possible that a more systematic approach to identifying such elements would allow more rapid or even more valuable insights; while Tor did prove to be an excellent project to center the investigation, it is entirely feasible a particular tool like Signal, or a field such as censorship circumvention, would be suited for this role as well. Regardless, applying these and future results gives some hope that the future of PETs can be defined not by beneficent experts, authorities, or platforms, but by empowering the users of these tools.

References

- [AAH13] Simurgh Aryan, Homa Aryan, and J Alex Halderman. “Internet Censorship in Iran: A First Look”. In: *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI 13)*. Aug. 2013 (cit. on pp. 10, 71).
- [ABEG13] Mashael AlSabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. “The Path Less Travelled: Overcoming Tor’s Bottlenecks with Traffic Splitting”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2013. DOI: 10.1302/3114-230258 (cit. on p. 10).
- [ABG+11] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey M Voelker. “DefenestraTor: Throwing Out Windows in Tor”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2011, pp. 134–154. DOI: 10.1007/978-3-642-22263-4_8 (cit. on p. 10).
- [ADS03] Alessandro Acquisti, Roger Dingledine, and Paul Syverson. “On the Economics of Anonymity”. In: *7th International Financial Cryptography Conference (FC)*. 2003. DOI: 10.1007/978-3-540-45126-6_7 (cit. on pp. 10, 36, 37, 59, 71).
- [AF11] Michel Jose Anzanello and Flavio Sanson Fogliatto. “Learning curve models and applications: Literature review and research directions”. In: *International Journal of Industrial Ergonomics* 41.5 (2011), pp. 573–583. DOI: 10.1016/j.ergon.2011.05.001 (cit. on pp. 82, 83).
- [AG13] Mashael AlSabah and Ian Goldberg. “PCTCP: Per-circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2013. DOI: 10.1145/2508859.2516715 (cit. on p. 10).
- [AG16] Mashael AlSabah and Ian Goldberg. “Performance and Security Improvements for Tor: A Survey”. In: *ACM Computing Surveys (CSUR)* 49.2 (2016), p. 32. DOI: 10.1145/2946802 (cit. on p. 14).

- [AM15] Jacob Appelbaum and Alec Muffett. *The ".onion" Special-Use Domain Name*. RFC 7686. Oct. 2015. DOI: 10.17487/RFC7686. URL: <https://www.rfc-editor.org/info/rfc7686> (cit. on p. 7).
- [ARK+18] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. "We Don't Need Another Hero? The Impact of "Heroes" on Software Development". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 245–253. DOI: 10.1145/3183519.3183549 (cit. on pp. 73, 74).
- [Aro24] Arushi Arora. "Privacy and Security Enhancements for Tor". PhD thesis. Purdue University, Apr. 2024. DOI: 10.25394/PGS.25653537.v1 (cit. on pp. 11, 12, 36, 95).
- [BBB+24] Enrico Bacis, Igor Bilogrevic, Robert Busa-Fekete, Asanka Herath, Antonio Sartori, and Umar Syed. "Assessing Web Fingerprinting Risk". In: *Companion Proceedings of the ACM on Web Conference 2024*. WWW '24. Singapore: Association for Computing Machinery, 2024, pp. 245–254. DOI: 10.1145/3589335.3648322 (cit. on p. 55).
- [BCH+14] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. "Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 257–268. DOI: 10.1145/2635868.2635880 (cit. on pp. 77, 91).
- [Ben14] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *arXiv preprint arXiv:1407.3561* (2014) (cit. on p. 51).
- [BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. "Off-the-Record Communication, or, Why Not To Use PGP". In: *Proceedings of the 2004 ACM workshop on Privacy in the Electronic Society*. 2004, pp. 77–84. DOI: 10.1145/1029179.1029200 (cit. on p. 46).
- [Bid16] Sam Biddle. "Apple Logs Your iMessage Contacts — and May Share Them With Police". In: *The Intercept* (Sept. 2016). URL: <https://theintercept.com/2016/09/28/apple-logs-your-imessage-contacts-and-may-share-them-with-police/> (visited on 05/2024) (cit. on p. 48).

- [Bid24] Sam Biddle. “This Undisclosed WhatsApp Vulnerability Lets Governments See Who You Message”. In: *The Intercept* (May 2024). URL: <https://theintercept.com/2024/05/22/whatsapp-security-vulnerability-meta-israel-palestine/> (visited on 06/2024) (cit. on p. 48).
- [BLMG21] Gabrielle Beck, Julia Len, Ian Miers, and Matthew Green. “Fuzzy Message Detection”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 1507–1528. DOI: 10.1145/3460120.3484545 (cit. on p. 51).
- [Boc23] Cecylia Bocovich. *Experimenting with Tor pluggable transports in Shadow*. Sept. 2023. URL: <https://forum.torproject.org/t/experimenting-with-tor-pluggable-transports-in-shadow/9117> (visited on 06/2024) (cit. on p. 95).
- [Bra24] Brave. *Brave Browser*. 2024. URL: <https://brave.com/> (visited on 05/2024) (cit. on p. 36).
- [BSBH19] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. “SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project”. In: *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*. MaLTesQuE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 7–12. DOI: 10.1145/3340482.3342742 (cit. on p. 80).
- [BSG11] Kevin S Bauer, Micah Sherr, and Dirk Grunwald. “ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation”. In: *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. 2011. DOI: 10.5555/2027999.2028006 (cit. on p. 13).
- [BW16] Armon Barton and Matthew Wright. “DeNASA: Destination-Naive AS-Awareness in Anonymous Communications”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2016.4* (2016), pp. 356–372. DOI: 10.1515/popets-2016-0044 (cit. on p. 10).
- [CA94] Albert T Corbett and John R Anderson. “Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge”. In: *User modeling and user-adapted interaction 4.4* (1994), pp. 253–278. DOI: 10.1007/BF01099821 (cit. on p. 83).

- [COA07] Jeremy Clark, P. C. van Oorschot, and Carlisle Adams. “Usability of Anonymous Web Browsing: An Examination of Tor Interfaces and Deployability”. In: *3rd Symposium on Usable Privacy and Security (SOUPS)*. 2007. DOI: 10.1145/1280680.1280687 (cit. on p. 10).
- [CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. “The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. CCS ’20*. 2020. DOI: 10.1145/3372297.3417887 (cit. on p. 49).
- [Cri23] Loredana Crisan. *Launching Default End-to-End Encryption on Messenger*. Meta. Dec. 2023. URL: <https://about.fb.com/news/2023/12/default-end-to-end-encryption-on-messenger/> (visited on 05/2024) (cit. on p. 48).
- [CS14] Bernd Conrad and Fatemeh Shirazi. “Analyzing the Effectiveness of DoS Attacks on Tor”. In: *7th International Conference on Security of Information and Networks*. 2014, p. 355. DOI: 10.1145/2659651.2659707 (cit. on p. 14).
- [Cur24] David Curry. *Signal Revenue & Usage Statistics (2024)*. Business of Apps. Jan. 2024. URL: <https://www.businessofapps.com/data/signal-statistics/> (visited on 04/2024) (cit. on pp. 2, 55, 67).
- [Dal13] Thomas M. Dalton. *AFFIDAVIT OF SPECIAL AGENT THOMAS M. DALTON 835 1682*. Accessible at https://web.archive.org/web/20140124073345/https://static.thecrimson.com/extras/2013/kim_affidavit.pdf. Dec. 2013 (cit. on p. 9).
- [Dan04] George Danezis. “The Traffic Analysis of Continuous-Time Mixes”. In: *Proceedings of Privacy Enhancing Technologies workshop (PET 2004)*. Vol. 3424. LNCS. May 2004, pp. 35–50. DOI: 10.1007/11423409_3 (cit. on pp. 8, 58).
- [DDB23] Hussein Darir, Geir E. Dullerud, and Nikita Borisov. “ProbFlow: Using Probabilistic Programming in Anonymous Communication Networks”. In: (2023). DOI: 10.14722/ndss.2023.24140 (cit. on pp. 11, 12, 36, 95).
- [DHK21] Claudia Diaz, Harry Halpin, and Aggelos Kiayias. *The Nym Network. The Next Generation of Privacy Infrastructure*. Tech. rep. Nym Technologies SA, Feb. 2021. URL: <https://nymtech.net/nym-whitepaper.pdf> (visited on 04/2024) (cit. on p. 50).

- [Din14] Roger Dingledine. *Tor security advisory: “relay early” traffic confirmation attack*. The Tor Project. 2014. URL: <https://blog.torproject.org/tor-security-advisory-relay-early-traffic-confirmation-attack/> (visited on 05/2024) (cit. on p. 9).
- [Dix24] Stacy Jo Dixon. *Most popular global mobile messenger apps as of January 2024, based on number of monthly active users*. Statista. Jan. 2024. URL: <https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/> (visited on 04/2024) (cit. on pp. 2, 46).
- [DLKS15] S. Dahal, Junghee Lee, Jungmin Kang, and Seokjoo Shin. “Analysis on End-to-End Node Selection Probability in Tor Network”. In: *2015 International Conference on Information Networking (ICOIN)*. Jan. 2015, pp. 46–50. DOI: 10.1109/icoin.2015.7057855 (cit. on p. 10).
- [DM06] Roger Dingledine and Nick Mathewson. “Anonymity Loves Company: Usability and the Network Effect”. In: *5th Workshop on the Economics of Information Security (WEIS)*. 2006 (cit. on pp. 10, 59).
- [DMMK18] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. “Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency—Choose Two”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 108–126. DOI: 10.1109/sp.2018.00011 (cit. on p. 9).
- [DMS+16] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. “A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes”. In: *IEEE Transactions on Software Engineering* 43.7 (2016), pp. 641–657. DOI: 10.1109/tse.2016.2616306 (cit. on pp. 76, 88).
- [DMS04] Roger Dingledine, Nick Mathewson, and Paul F Syverson. “Tor: The Second-Generation Onion Router”. In: *USENIX Security Symposium*. Vol. 4. 2004, pp. 303–320 (cit. on pp. 5, 58).
- [DRPW20] Thien-Nam Dinh, Florentin Rochet, Olivier Pereira, and Dan S Wallach. “Scaling Up Anonymous Communication with Efficient Nanopayment Channels”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2020.3* (2020), pp. 175–203. DOI: 10.2478/popets-2020-0048 (cit. on p. 10).
- [DSC+22] Hussein Darir, Hussein Sibai, Chin-Yu Cheng, Nikita Borisov, Geir Dullerud, and Sayan Mitra. “Mleflow: Learning from History to Improve Load balancing in Tor”. In: *Proceedings on Privacy Enhancing Technologies* (2022). DOI: 10.2478/popets-2022-0005 (cit. on pp. 11, 12, 36, 95).

- [DT23] Christoph Döpmann and Florian Tschorsch. “Modeling Tor Network Growth by Extrapolating Consensus Data”. In: *Proceedings of the 18th International Conference on Availability, Reliability and Security*. ARES ’23. New York, NY, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3600160.3600164 (cit. on pp. 11, 12, 36, 95).
- [Eck10] Peter Eckersley. “How Unique Is Your Web Browser?” In: *Privacy Enhancing Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–18. DOI: 10.1007/978-3-642-14527-8_1 (cit. on p. 55).
- [EDG14] Tariq Elahi, George Danezis, and Ian Goldberg. “PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks”. In: *ACM Conference on Computer and Communications Security (CCS)*. See also `git : / / git - crysp . uwaterloo . ca / privex`. 2014. DOI: 10.1145/2660267.2660280 (cit. on p. 14).
- [Egh20] Nadia Eghbal. *Working in Public: the Making and Maintenance of Open Source Software*. Stripe Press, 2020 (cit. on p. 74).
- [EGS21] Peter Elkind, Jack Gillum, and Craig Silverman. “How Facebook Undermines Privacy Protections for Its 2 Billion WhatsApp Users”. In: *ProPublica* (Sept. 2021). URL: <https://www.propublica.org/article/how-facebook-undermines-privacy-protections-for-its-2-billion-whatsapp-users> (visited on 05/2024) (cit. on p. 48).
- [EHM17] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. “Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols”. In: *European Workshop on Usable Security*. 2017. DOI: 10.14722/eurosec.2017.23016 (cit. on p. 3).
- [Esr24] Esra’a al Shafei. “The dangers of metadata in messengers”. In: (Apr. 2024). URL: <https://simplex.chat/blog/20240416-dangers-of-metadata-in-messengers.html> (visited on 04/2024) (cit. on pp. 45, 95).
- [FCV+21] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. “Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study”. In: *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, Aug. 2021, pp. 597–616 (cit. on pp. 74, 94, 96).

- [Fed12] Federal Bureau of Investigation. *Timeline Correlation: Hammond and Anarchaos*. Accessible at <https://www.documentcloud.org/documents/1342115-timeline-correlation-jeremy-hammond-and-anarchaos>. 2012 (cit. on p. 9).
- [FGK+10] Benjamin Fabian, Florian Goertz, Steffen Kunz, Sebastian Müller, and Mathias Nitzsche. “Privately Waiting — A Usability Analysis of the Tor Anonymity Network”. In: *Sustainable e-Business Management*. Aug. 2010. DOI: 10.1007/978-3-642-15141-5_6 (cit. on pp. 10, 37, 59, 71).
- [Fle09] Gregory Fleischer. *Attacking Tor at the Application Layer*. DEF CON 17. 2009. URL: <https://www.youtube.com/watch?v=nEUPdUniVxQ> (visited on 04/2024) (cit. on p. 46).
- [FMJS17] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. “Distributed Measurement with Private Set-Union Cardinality”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134034 (cit. on p. 14).
- [Gab19] Gaba. *Mozilla Research Call: Tune up Tor for Integration and Scale*. The Tor Project. May 2019. URL: <https://blog.torproject.org/mozilla-research-call-tune-tor-integration-and-scale> (visited on 05/2024) (cit. on p. 37).
- [GH12] Deepika Gopal and Nadia Heninger. “Torchestra: Reducing Interactive Traffic Delays over Tor”. In: *ACM Workshop on Privacy in the Electronic Society (WPES)*. 2012. DOI: 10.1145/2381966.2381972 (cit. on p. 10).
- [GHC14] Major Garrett, Michael Hayden, and David Cole. “Debate: The Price of Privacy. Re-Evaluating the NSA”. Johns Hopkins Foreign Affairs Symposium. 2014. URL: <https://www.youtube.com/watch?v=kV2HDM86XgI> (visited on 04/2024) (cit. on p. 45).
- [GJH13] John Geddes, Rob Jansen, and Nicholas Hopper. “How Low Can You Go: Balancing Performance with Anonymity in Tor”. In: *13th Privacy Enhancing Technologies Symposium*. 2013, pp. 164–184. DOI: 10.1007/978-3-642-39077-7_9 (cit. on p. 10).
- [GJH14] John Geddes, Rob Jansen, and Nicholas Hopper. “IMUX: Managing Tor Connections from Two to Infinity, and Beyond”. In: *ACM Workshop on Privacy in the Electronic Society (WPES)*. 2014, pp. 181–190. DOI: 10.1145/2665943.2665948 (cit. on p. 10).

- [GRS99] David Goldschlag, Michael Reed, and Paul Syverson. “Onion Routing”. In: *Communications of the ACM* 42.2 (1999), pp. 39–41. DOI: 10.1145/293411.293443 (cit. on p. 5).
- [GSH16] John Geddes, Mike Schliep, and Nicholas Hopper. “ABRA CADABRA: Magically Increasing Network Utilization in Tor by Avoiding Bottlenecks”. In: *15th ACM Workshop on Privacy in the Electronic Society*. 2016, pp. 165–176. DOI: 10.1145/2994620.2994630 (cit. on p. 10).
- [GT18] Kiran Garimella and Gareth Tyson. “WhatsApp, Doc? A First Look at WhatsApp Public Group Data”. In: *Proceedings of the Twelfth International AAAI Conference on Web and Social Media (ICWSM 2018)*. Vol. 12. 1. 2018. DOI: 10.1609/icwsm.v12i1.14989 (cit. on p. 64).
- [HGK+19] Mohammadreza Hazhirpasand, Mohammad Ghafari, Stefan Krüger, Eric Bodden, and Oscar Nierstrasz. “The Impact of Developer Experience in Using Java Cryptography”. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 1–6. DOI: 10.1109/ESEM.2019.8870184 (cit. on pp. 77, 91).
- [Hoa10] Graydon Hoare. “Project Servo: Technology from the past come to save the future from itself”. In: *Mozilla Annual Summit*. July 2010 (cit. on p. 77).
- [Hoe71] Paul G. Hoel. *Introduction to Mathematical Statistics*. 4th. New York: Wiley, 1971. Chap. Applications of the t Distribution, pp. 261–262. ISBN: 0471403652 (cit. on p. 33).
- [Hop14] Nicholas Hopper. “Challenges in protecting Tor hidden services from botnet abuse”. In: *Financial Cryptography and Data Security (FC)*. 2014, pp. 316–325. DOI: 10.1007/978-3-662-45472-5_21 (cit. on p. 14).
- [HSWM19] Hans Hanley, Yixin Sun, Sameer Wagh, and Prateek Mittal. “DPSelect: A Differential Privacy Based Guard Relay Selection Algorithm for Tor”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2019.2* (2019), pp. 166–186. DOI: 10.2478/popets-2019-0025 (cit. on p. 10).
- [IAW19] Mohsen Imani, Mehrdad Amirabadi, and Matthew Wright. “Modified Relay Selection and Circuit Selection for Faster Tor”. In: *IET Communications* 13.17 (2019), pp. 2723–2734. DOI: 10.1049/iet-com.2018.5591 (cit. on p. 10).
- [IBW18] Mohsen Imani, Armon Barton, and Matthew Wright. “Guard Sets in Tor using AS Relationships”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2018.1* (2018), pp. 145–165. DOI: 10.1515/popets-2018-0008 (cit. on p. 10).

- [JBHD12] Rob Jansen, Kevin Bauer, Nicholas Hopper, and Roger Dingledine. “Methodically Modeling the Tor Network”. In: *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. 2012 (cit. on pp. 11, 14, 21, 25).
- [JGW+14] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. “Never Been KIST: Tor’s Congestion Management Blossoms with Kernel-Informed Socket Transport”. In: *USENIX Security Symposium (USENIX-Sec)*. 2014. DOI: 10.5555/2671225.2671234 (cit. on pp. 10, 13).
- [JH12] Rob Jansen and Nicholas Hopper. “Shadow: Running Tor in a Box for Accurate and Efficient Experimentation”. In: *Network and Distributed System Security Symposium (NDSS)*. See also <https://shadow.github.io>. Feb. 2012. DOI: 10.21236/ada559181 (cit. on pp. 12, 13, 24, 37).
- [JHK10] Rob Jansen, Nicholas Hopper, and Yongdae Kim. “Recruiting New Tor Relays with BRAIDS”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2010. DOI: 10.1145/1866307.1866344 (cit. on pp. 10, 13).
- [JJ16] Rob Jansen and Aaron Johnson. “Safely Measuring Tor”. In: *ACM Conference on Computer and Communications Security (CCS)*. See also <https://github.com/privcount>. 2016. DOI: 10.1145/2976749.2978310 (cit. on pp. 14, 18, 22).
- [JJH+17] Aaron Johnson, Rob Jansen, Nicholas Hopper, Aaron Segal, and Paul Syverson. “PeerFlow: Secure Load Balancing in Tor”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2017.2* (2017), pp. 74–94. DOI: 10.1515/popets-2017-0017 (cit. on pp. 10, 14).
- [JJJ+17] Aaron Johnson, Rob Jansen, Aaron D Jaggard, Joan Feigenbaum, and Paul Syverson. “Avoiding The Man on the Wire: Improving Tor’s Security with Trust-Aware Path Selection”. In: *Network and Distributed System Security Symposium (NDSS)*. 2017. DOI: 10.14722/ndss.2017.23307 (cit. on p. 10).
- [JJS13] Rob Jansen, Aaron Johnson, and Paul Syverson. “LIRA: Lightweight Incentivized Routing for Anonymity”. In: *Network and Distributed System Security Symposium (NDSS)*. 2013 (cit. on pp. 10, 14).
- [JNW22] Rob Jansen, Jim Newsome, and Ryan Wails. “Co-opting Linux Processes for High-Performance Network Simulation”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. July 2022, pp. 327–350 (cit. on pp. 13, 66).
- [JSH12] Rob Jansen, Paul F Syverson, and Nicholas Hopper. “Throttling Tor Bandwidth Parasites”. In: *USENIX Security Symposium (USENIX-Sec)*. 2012 (cit. on pp. 10, 37).

- [JTG+18] Rob Jansen, Matthew Traudt, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. “KIST: Kernel-Informed Socket Transport for Tor”. In: *ACM Transactions on Privacy and Security (TOPS)* 22.1 (Dec. 2018), 3:1–3:37. DOI: 10.1145/3278121 (cit. on pp. 10, 11, 13, 37).
- [JTG21] Rob Jansen, Justin Tracey, and Ian Goldberg. “Once is Never Enough: Foundations for Sound Statistical Inference in Tor Network Experimentation”. In: *30th USENIX Security Symposium (Sec)*. See also <https://neverenough-sec2021.github.io>. 2021 (cit. on p. iv).
- [JTH18] Rob Jansen, Matthew Traudt, and Nicholas Hopper. “Privacy-Preserving Dynamic Learning of Tor Network Traffic”. In: *ACM Conference on Computer and Communications Security (CCS)*. CCS ’18. See also <https://tmodel-ccs2018.github.io>. 2018, pp. 1944–1961. DOI: 10.1145/3243734.3243815 (cit. on pp. 12–14, 18, 20–23, 25–27, 29).
- [JTJS14] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. “The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network”. In: *Network and Distributed System Security Symposium (NDSS)*. 2014. DOI: 10.14722/ndss.2014.23288 (cit. on p. 14).
- [JVS19] Rob Jansen, Tavish Vaidya, and Micah Sherr. “Point Break: A Study of Bandwidth Denial-of-Service Attacks against Tor”. In: *USENIX Security Symposium (USENIX-Sec)*. 2019. DOI: 10.5555/3361338.3361465 (cit. on p. 14).
- [JW23] Rob Jansen and Ryan Wails. “Data-explainable website fingerprinting with network simulation”. In: *Proceedings on Privacy Enhancing Technologies* (2023). DOI: 10.56553/popets-2023-0125 (cit. on pp. 11, 12, 36, 95).
- [KBS+19] Christiane Kuhn, Martin Beck, Stefan Schiffner, Eduard Jorswieck, and Thorsten Strufe. “On Privacy Notions in Anonymous Communication”. In: *Proceedings on Privacy Enhancing Technologies* 2 (2019), pp. 105–125. DOI: 10.2478/popets-2019-0022 (cit. on p. 47).
- [KCU+19] Kiran K, Saurabh S Chalke, Mohammad Usman, P Deepa Shenoy, and Venugopal K R. “Anonymity and Performance Analysis of Stream Isolation in Tor Network”. In: *International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2019. DOI: 10.1109/icccnt45670.2019.8944443 (cit. on p. 10).
- [key24] keys.openpgp.org. *Stats*. May 2024. URL: <https://keys.openpgp.org/about/stats> (visited on 05/2024) (cit. on p. 2).

- [KMG20] Chelsea H. Komlo, Nick Mathewson, and Ian Goldberg. “Walking Onions: Scaling Anonymity Networks while Protecting Users”. In: *USENIX Security Symposium (USENIX-Sec)*. 2020 (cit. on p. 37).
- [KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. “Automatic Identification of Bug-Introducing Changes”. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. IEEE. 2006, pp. 81–90. DOI: 10.1109/ase.2006.23 (cit. on pp. 76, 88).
- [LCBS23] Daniela Lopes, Daniel Castro, Diogo Barradas, and Nuno Santos. “TIGER: Tor Traffic Generator for Realistic Experiments”. In: *Proceedings of the 22nd Workshop on Privacy in the Electronic Society*. 2023, pp. 147–152. DOI: 10.1145/3603216.3624960 (cit. on p. 13).
- [Lew18] Sarah Jamie Lewis. *Cwtch: Privacy Preserving Infrastructure for Asynchronous, Decentralized, Multi-Party and Metadata Resistant Applications*. Tech. rep. 2018. URL: <https://cwtch.im/cwtch.pdf> (visited on 04/2024) (cit. on p. 50).
- [LFM+17] Linda Lee, David Fifield, Nathan Malkin, Ganesh Iyer, Serge Egelman, and David Wagner. “A Usability Evaluation of Tor Launcher”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2017* (July 2017). DOI: 10.1515/popets-2017-0030 (cit. on p. 10).
- [Lin23] Linux Kernel authors. *proc(5) File Formats Manual*. 6.03. Linux man-pages. Feb. 2023 (cit. on p. 66).
- [LLW+17] Zhuotao Liu, Yushan Liu, Philipp Winter, Prateek Mittal, and Yih-Chun Hu. “TorPo-lice: Towards Enforcing Service-Defined Access Policies for Anonymous Communication in the Tor Network”. In: *International Conference on Network Protocols*. 2017. DOI: 10.1109/icnp.2017.8117564 (cit. on p. 10).
- [LSB08] Ashvin Lakshmikantha, R. Srikant, and Carolyn Beck. “Impact of File Arrivals and Departures on Buffer Sizing in Core Routers”. In: *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. May 2008. DOI: 10.1109/infocom.2008.26 (cit. on p. 19).
- [LSL16] Dong Lin, Micah Sherr, and Boon Thau Loo. “Scalable and Anonymous Group Communication with MTor”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2016.2* (2016), pp. 22–39. DOI: 10.1515/popets-2016-0003 (cit. on p. 10).
- [Lun18] Joshua Lund. *Technology preview: Sealed sender for Signal*. Signal Technology Foundation. Oct. 2018. URL: <https://signal.org/blog/sealed-sender/> (visited on 04/2024) (cit. on p. 49).

- [MAB+10] Pere Manils, Chaabane Abdelberri, Stevens Le Blond, Mohamed Ali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. *Compromising Tor anonymity exploiting P2P information leakage*. Tech. rep. inria-00471556. 2010. URL: <https://inria.hal.science/inria-00471556/en/> (visited on 04/2024) (cit. on p. 46).
- [MAEP20] Asya Mitseva, Marharyta Aleksandrova, Thomas Engel, and Andriy Panchenko. “Security and Performance Implications of BGP Rerouting-Resistant Guard Selection Algorithms for Tor”. In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. 2020. DOI: 10.1016/j.cose.2023.103374 (cit. on p. 10).
- [Mar14] Moxie Marlinspike. *Contributing to Signal Android: Development Ideology*. Open Whisper Systems. June 2014. URL: <https://github.com/signalapp/Signal-Android/blob/main/CONTRIBUTING.md> (visited on 05/2024) (cit. on p. 2).
- [Mar15] Moxie Marlinspike. *GPG And Me*. Feb. 2015. URL: <https://moxie.org/2015/02/24/gpg-and-me.html> (visited on 05/2024) (cit. on p. 2).
- [Mar17] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Open Whisper Systems. Sept. 2017. URL: <https://signal.org/blog/private-contact-discovery/> (visited on 04/2024) (cit. on p. 49).
- [Mar19] Moxie Marlinspike. “The Ecosystem is Moving”. 36th Chaos Communication Congress. 2019. URL: https://media.ccc.de/v/36c3-11086-the_ecosystem_is_moving#t=2034 (visited on 04/2024) (cit. on pp. 45, 49, 59, 95).
- [Mat22] Nick Mathewson. *Arti 1.0.0 is released: Our Rust Tor implementation is ready for production use*. The Tor Project. Sept. 2022. URL: https://blog.torproject.org/arti_100_released/ (visited on 05/2024) (cit. on pp. 75, 94).
- [Met23] Meta Platforms, Inc. *Government Requests for User Data*. 2023. URL: <https://transparency.meta.com/reports/government-data-requests/> (visited on 05/2024) (cit. on p. 48).
- [Mic21] Pol Micolau Cos. “Simulació de la xarxa anònima Tor”. BA thesis. 2021. URL: <https://ddd.uab.cat/record/248472> (visited on 05/2024) (cit. on pp. 11, 12, 36, 95).

- [MJ15] Andrew Miller and Rob Jansen. “Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications”. In: *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. 2015. DOI: 10.5555/2831120.2831127 (cit. on p. 13).
- [MK14] Nicholas D. Matsakis and Felix S. Klock II. “The Rust Language”. In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT ’14. Portland, Oregon, USA: ACM, 2014, pp. 103–104. DOI: 10.1145/2663171.2663188 (cit. on p. 74).
- [MKA+21] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Daniel Roche, and Eric Wustrow. “Improving Signal’s Sealed Sender”. In: *Network and Distributed Systems Security (NDSS) Symposium*. 2021. DOI: 10.14722/ndss.2021.23180 (cit. on pp. 45, 57, 61, 95).
- [Mol23] The Molly project. *Molly*. 2023. URL: <https://molly.im> (visited on 04/2024) (cit. on pp. 45, 95).
- [Moz19] Mozilla. *Mozilla Research Grants 2019H1*. Call for Proposals. Archived at <https://web.archive.org/web/20200523195142/https://mozilla-research.forms.fm/mozilla-research-grants-2019h1/forms/6510>. 2019. URL: <https://mozilla-research.forms.fm/mozilla-research-grants-2019h1/forms/6510> (visited on 05/2020) (cit. on p. 36).
- [Moz20] Mozilla. *Oxidation*. 2020. URL: <https://wiki.mozilla.org/Oxidation> (visited on 05/2024) (cit. on pp. 75, 77).
- [Moz21] Mozilla Foundation. “WhatsApp”. In: *Privacy Not Included*. Sept. 2021. URL: <https://foundation.mozilla.org/en/privacynotincluded/whatsapp/> (visited on 04/2024) (cit. on p. 48).
- [Moz24] Mozilla. *Firefox Public Data Report*. May 2024. URL: <https://data.firefox.com/dashboard/user-activity> (visited on 05/2024) (cit. on p. 37).
- [MS17] Akshaya Mani and Micah Sherr. “HisTor ϵ : Differentially Private and Robust Statistics Collection for Tor”. In: *Network and Distributed System Security Symposium (NDSS)*. 2017. DOI: 10.14722/ndss.2017.23411 (cit. on p. 14).
- [MSM+13] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. “When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits”. In: *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE. 2013, pp. 65–74. DOI: 10.1109/ESEM.2013.19 (cit. on pp. 76, 81).

- [MW00] A. Mockus and D. M. Weiss. “Predicting Risk of Software Changes”. In: *Bell Labs Technical Journal* 5.2 (Apr. 2000), pp. 169–180. DOI: 10.1002/bltj.2229 (cit. on pp. 77, 91).
- [MWJ+18] Akshaya Mani, T Wilson-Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. “Understanding Tor Usage with Privacy-Preserving Measurement”. In: *18th ACM Internet Measurement Conference (IMC)*. See also <https://torusage-imc2018.github.io>. 2018. DOI: 10.1145/3278532.3278549 (cit. on pp. 14, 18).
- [MWS11] W. Brad Moore, Chris Wacek, and Micah Sherr. “Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race with Tortoise”. In: *Annual Computer Security Applications Conference (ACSAC)*. 2011. DOI: 10.1145/2076732.2076762 (cit. on p. 10).
- [NCC12] Greg Norcie, Kelly Caine, and L. Jean Camp. “Eliminating Stop-Points in the Installation and Use of Anonymity Systems: a Usability Evaluation of the Tor Browser Bundle”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2012. DOI: 10.1007/978-3-642-31680-7 (cit. on p. 10).
- [NDW10] Tsuen-Wan “Johnny” Ngan, Roger Dingledine, and Dan S. Wallach. “Building Incentives into Tor”. In: *Financial Cryptography and Data Security (FC)*. 2010. DOI: 10.1007/978-3-642-14577-3_19 (cit. on p. 13).
- [NR81] Allen Newell and P Rosenbloom. “Mechanisms of Skill Acquisition and the Law of Practice”. In: *Cognitive skills and their acquisition* (1981) (cit. on p. 83).
- [NZHZ07] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. “Predicting Vulnerable Software Components”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. Alexandria, Virginia, USA: ACM, 2007, pp. 529–540. DOI: 10.1145/1315245.1315311 (cit. on p. 77).
- [OAD22] Lennart Oldenburg, Gunes Acar, and Claudia Diaz. “From “Onion Not Found” to Guard Discovery”. In: *Proceedings on Privacy Enhancing Technologies* (2022). DOI: 10.2478/popets-2022-0026 (cit. on pp. 11, 12, 36, 95).
- [OLe19] Jim O’Leary. *Technology Preview: Signal Private Group System*. Signal Technology Foundation. Dec. 2019. URL: <https://signal.org/blog/signal-private-group-system/> (visited on 04/2024) (cit. on p. 49).
- [Ope21] Open Privacy. *niwl*. 2021. URL: <https://git.openprivacy.ca/openprivacy/niwl> (visited on 04/2024) (cit. on p. 51).

- [PAKE23] Sebastian Pahl, Florian Adamsky, Daniel Kaiser, and Thomas Engel. “Examining the Hydra: Simultaneously Shared Links in Tor and the Effects on its Performance”. In: *Proceedings on Privacy Enhancing Technologies* (2023). DOI: 10.56553/popets-2023-0081 (cit. on pp. 11, 12, 36, 95).
- [PDS+15] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. “VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. CCS ’15*. Denver, Colorado, USA: ACM, 2015, pp. 426–437. DOI: 10.1145/2810103.2813604 (cit. on p. 77).
- [Pla24] Google Play. *Firefox Fast & Private Browser*. Apr. 2024. URL: <https://play.google.com/store/apps/details?id=org.mozilla.firefox> (visited on 05/2024) (cit. on p. 2).
- [PSG16] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. “More Common Than You Think: An In-depth Study of Casual Contributors”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 112–123. DOI: 10.1109/SANER.2016.68 (cit. on p. 74).
- [RD11] Foyzur Rahman and Premkumar Devanbu. “Ownership, Experience and Defects: A Fine-grained Study of Authorship”. In: *Proceedings of the 33rd International Conference on Software Engineering. ICSE ’11*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 491–500. DOI: 10.1145/1985793.1985860 (cit. on p. 77).
- [RE22] Florentin Rochet and Tariq Ehsan Elahi. “Towards Flexible Anonymous Networks”. In: *ArXiv abs/2203.03764* (2022) (cit. on pp. 11, 12, 36, 95).
- [RM10] Filippo Ricca and Alessandro Marchetto. “Are Heroes Common in FLOSS Projects?” In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM ’10*. Bolzano-Bozen, Italy: Association for Computing Machinery, 2010. DOI: 10.1145/1852786.1852856 (cit. on p. 74).
- [RP17] Florentin Rochet and Olivier Pereira. “Waterfilling: Balancing the Tor network with maximum diversity”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs) 2017.2* (2017), pp. 4–22. DOI: 10.1515/popets-2017-0013 (cit. on p. 10).

- [RP18] Florentin Rochet and Olivier Pereira. “Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2018.2 (2018), pp. 27–46. DOI: 10.1515/popets-2018-0011 (cit. on p. 14).
- [RWJ+20] Florentin Rochet, Ryan Wails, Aaron Johnson, Prateek Mittal, and Olivier Pereira. “CLAPS: Client-Location-Aware Path Selection in Tor”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2020. DOI: 10.1145/3372297.3417279 (cit. on p. 10).
- [Sch13] Bruce Schneier. “Attacking Tor: how the NSA targets users’ online anonymity”. In: *The Guardian* (Oct. 2013). URL: <https://www.theguardian.com/world/2013/oct/04/tor-attacks-nsa-users-online-anonymity> (visited on 05/2024) (cit. on p. 8).
- [SDW15] Fatemeh Shirazi, Claudia Diaz, and Joss Wright. “Towards Measuring Resilience in Anonymous Communication Networks”. In: *14th ACM Workshop on Privacy in the Electronic Society*. 2015, pp. 95–99. DOI: 10.1145/2808138.2808152 (cit. on p. 10).
- [SG24] Sajin Sasy and Ian Goldberg. “SoK: Metadata-Protecting Communication Systems”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* (2024). DOI: 10.56553/popets-2024-0030 (cit. on p. 47).
- [SGD15] Fatemeh Shirazi, Matthias Goehring, and Claudia Diaz. “Tor Experimentation Tools”. In: *International Workshop on Privacy Engineering (IWPE)*. 2015. DOI: 10.1109/SPW.2015.20 (cit. on pp. 11, 13, 24).
- [Sin14] Sukhbir Singh. “Large-Scale Emulation of Anonymous Communication Networks”. MA thesis. University of Waterloo, 2014. URL: <https://hdl.handle.net/10012/8642> (cit. on p. 13).
- [SKBP23] Theodor Schnitzler, Katharina Kohls, Evangelos Bitsikas, and Christina Pöpper. “Hope of Delivery: Extracting User Locations From Mobile Instant Messengers”. In: *Network and Distributed Systems Security (NDSS) Symposium 2023*. Feb. 2023. DOI: 10.14722/ndss.2023.23188 (cit. on p. 65).
- [Smi19] Al Smith. *Tor Bug Smash Fund*. The Tor Project. Aug. 2019. URL: <https://blog.torproject.org/tors-bug-smash-fund-help-tor-smash-all-bugs/> (visited on 05/2024) (cit. on p. 73).
- [Sol07] Daniel J Solove. ““I’ve Got Nothing to Hide” and Other Misunderstandings of Privacy”. In: *San Diego L. Rev.* 44 (2007), p. 745 (cit. on p. 1).

- [SPSH23] Anika Seufert, Fabian Poignée, Michael Seufert, and Tobias Hoßfeld. “Share and Multiply: Modeling Communication and Generated Traffic in Private WhatsApp Groups”. In: *IEEE Access* 11 (2023), pp. 25401–25414. DOI: 10.1109/access.2023.3254913 (cit. on p. 63).
- [SSHT15] Michael Seufert, Anika Schwind, Tobias Hoßfeld, and Phuoc Tran-Gia. “Analysis of Group-Based Communication in WhatsApp”. In: *Mobile Networks and Management: 7th International Conference, MONAMI 2015, Santander, Spain, September 16-18, 2015, Revised Selected Papers 7*. Springer, 2015, pp. 225–238. DOI: 10.1007/978-3-319-26925-2_17 (cit. on p. 63).
- [Sta24a] Statcounter GlobalStats. *Mobile & Tablet Android Version Market Share Worldwide*. June 2024. URL: <https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide> (visited on 06/2024) (cit. on p. 57).
- [Sta24b] Statcounter GlobalStats. *Mobile Browser Markey Share Worldwide*. May 2024. URL: <https://gs.statcounter.com/browser-market-share/mobile/worldwide> (visited on 05/2024) (cit. on p. 2).
- [SWCG13] Igor Steinmacher, Igor Wiese, Ana Paula Chaves, and Marco Aurélio Gerosa. “Why do newcomers abandon open source software projects?” In: *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 2013, pp. 25–32. DOI: 10.1109/CHASE.2013.6614728 (cit. on p. 74).
- [ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When Do Changes Induce Fixes?” In: *MSR ’05: Proceedings of the 2005 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2005, pp. 1–5. DOI: 10.1145/1082983.1083147 (cit. on pp. 75, 126).
- [TG10] Can Tang and Ian Goldberg. “An Improved Algorithm for Tor Circuit Scheduling”. In: *17th ACM Conference on Computer and Communications Security (CCS)*. 2010. DOI: 10.1145/1866307.1866345 (cit. on pp. 10, 11).
- [TG23] Justin Tracey and Ian Goldberg. “Grading on a Curve: How Rust can Facilitate New Contributors while Decreasing Vulnerabilities”. In: *2023 IEEE Secure Development Conference (SecDev)*. IEEE, 2023, pp. 26–36. DOI: 10.1109/secdev56634.2023.00016 (cit. on p. iv).
- [TJG18] Justin Tracey, Rob Jansen, and Ian Goldberg. “High Performance Tor Experimentation from the Magic of Dynamic ELF’s”. In: *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. 2018. DOI: 10.5555/3307412.3307422 (cit. on p. 13).

- [Tor] The Tor Project. *Path selection and constraints*. URL: <https://spec.torproject.org/path-spec/path-selection-constraints.html> (visited on 06/2024) (cit. on p. 56).
- [Tor23] The Tor Project. *Reproducible Metrics*. 2023. URL: <https://metrics.torproject.org/reproducible-metrics.html#performance> (visited on 05/2024) (cit. on pp. 26–29).
- [Tor24] The Tor Project. *Tor Metrics Portal*. 2024. URL: <https://metrics.torproject.org> (visited on 05/2024) (cit. on pp. 10, 15, 20, 23, 26, 55).
- [Ubu24] Ubuntu Keyserver. *Hockey puck OpenPGP Keyserver Statistics*. May 2024. URL: <https://keyserver.ubuntu.com/pks/lookup?op=stats> (visited on 05/2024) (cit. on p. 2).
- [Ung17] Nik Unger. *NetMirage*. 2017. URL: <https://crysp.uwaterloo.ca/software/netmirage/> (visited on 04/2020) (cit. on p. 13).
- [Uni20] United Nations. *Freedom of Information*. <https://www.un.org/ruleoflaw/thematic-areas/governance/freedom-of-information>. Jan. 2020 (cit. on p. 36).
- [Uns23] Unsafe Code Guidelines working group. *Unsafe Code Guidelines Reference*. 2023. URL: <https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#soundness-of-code--of-a-library> (visited on 06/2023) (cit. on p. 87).
- [Von21] Manuel Vonau. “Signal updates open-source server code after it failed to for nearly a year”. In: *Android Police* (Apr. 2021). URL: <https://www.androidpolice.com/2021/04/06/it-looks-like-signal-isnt-as-open-source-as-you-thought-it-was-anymore/> (visited on 07/2024) (cit. on p. 51).
- [VYW+03] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. “Scalability and Accuracy in a Large-Scale Network Emulator”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 271–284. DOI: 10.1145/1060289.1060315 (cit. on p. 13).
- [Wha21a] WhatsApp LLC. *Answering your questions about WhatsApp’s January 2021 Privacy Policy update*. 2021. URL: <https://faq.whatsapp.com/595724415641642> (visited on 04/2024) (cit. on p. 48).
- [Wha21b] WhatsApp LLC. *WhatsApp Privacy Policy*. Jan. 2021. URL: <https://www.whatsapp.com/legal/privacy-policy> (visited on 04/2024) (cit. on p. 48).

- [WHH22] Ethan Witwer, James K. Holland, and Nicholas Hopper. “Padding-only Defenses Add Delay in Tor”. In: *Proceedings of the 21st Workshop on Privacy in the Electronic Society*. WPES’22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 29–33. DOI: 10.1145/3559613.3563207 (cit. on pp. 11, 12, 36, 95).
- [WL23] Meredith Whittaker and Joshua Lund. *Privacy is Priceless, but Signal is Expensive*. Signal Technology Foundation. Nov. 2023. URL: <https://signal.org/blog/signal-is-expensive/> (visited on 04/2024) (cit. on p. 65).
- [WS08] Chadd Williams and Jaime Spacco. “SZZ Revisited: Verifying When Changes Induce Fixes”. In: *Proceedings of the 2008 workshop on Defects in large software systems (DEFACTS 2008)*. Seattle, Washington: Association for Computing Machinery, 2008, pp. 32–36. DOI: 10.1145/1390817.1390826 (cit. on pp. 76, 88).
- [WSS14] James Walden, Jeff Stuckman, and Riccardo Scandariato. “Predicting Vulnerable Components: Software Metrics vs Text Mining”. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. Nov. 2014, pp. 23–33. DOI: 10.1109/ISSRE.2014.32 (cit. on p. 77).
- [WT99] Alma Whitten and J Doug Tygar. “Why Johnny Can’t Encrypt: A Usability Evaluation of PGP 5.0.” In: *USENIX security symposium*. Vol. 348. 1999, pp. 169–184. DOI: 10.5555/1251421.1251435 (cit. on p. 2).
- [WTBS13] Chris Wacek, Henry Tan, Kevin Bauer, and Micah Sherr. “An Empirical Evaluation of Relay Selection in Tor”. In: *Network and Distributed System Security Symposium (NDSS)*. 2013 (cit. on p. 10).
- [Wyd23] Ron Wyden. *Letter to Attorney General Merrick B. Garland*. Dec. 2023. URL: https://www.wyden.senate.gov/imo/media/doc/wyden_smartphone_push_notification_surveillance_letter.pdf (visited on 06/2024) (cit. on p. 58).
- [YL15a] Lei Yang and Fengjun Li. “Enhancing Traffic Analysis Resistance for Tor Hidden Services with Multipath Routing”. In: *International Conference on Security and Privacy in Communication Systems*. 2015, pp. 367–384. DOI: 10.1109/cns.2015.7346915 (cit. on p. 10).
- [YL15b] Lei Yang and Fengjun Li. “mTor: A Multipath Tor Routing Beyond Bandwidth Throttling”. In: *2015 IEEE Conference on Communications and Network Security (CNS)*. Sept. 2015, pp. 479–487. DOI: 10.1109/cns.2015.7346860 (cit. on p. 10).

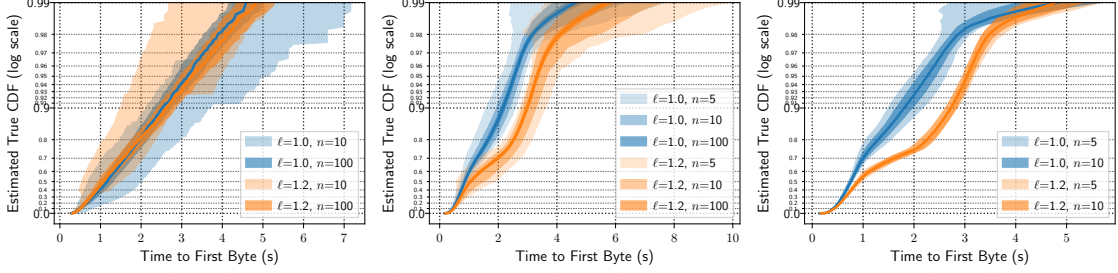
- [You17] Kirsty Young. *Sheryl Sandberg*. interview with. BBC. Aug. 2017. URL: <https://www.bbc.co.uk/programmes/b08z9b81> (visited on 04/2024) (cit. on p. 48).
- [ZMW+20] Pengxiong Zhu, Keyu Man, Zhongjie Wang, Zhiyun Qian, Roya Ensafi, J. Alex Halderman, and Haixin Duan. “Characterizing Transnational Internet Performance and the Great Bottleneck of China”. In: *Measurement and Analysis of Computing Systems* 4.1 (2020). DOI: 10.1145/3379479 (cit. on p. 19).

APPENDICES

Appendix A

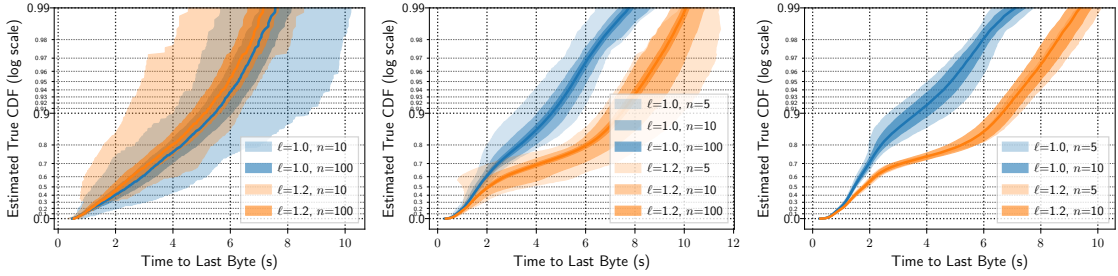
Additional Performance Results

We supplement the analysis done in Section 3.6.3 with additional plots of performance metrics measured across our 420 simulations following our statistical inference methodology from Section 3.5. Our supplemental results include the transfer times measured by the performance benchmarking clients: the time to first byte across all transfer sizes are shown in Figure A.1, while the time to last byte of 50 KiB and 5 MiB transfers are shown in Figure A.2 and Figure A.3, respectively. The total Tor network goodput (summed over all relays) is shown in Figure A.4.



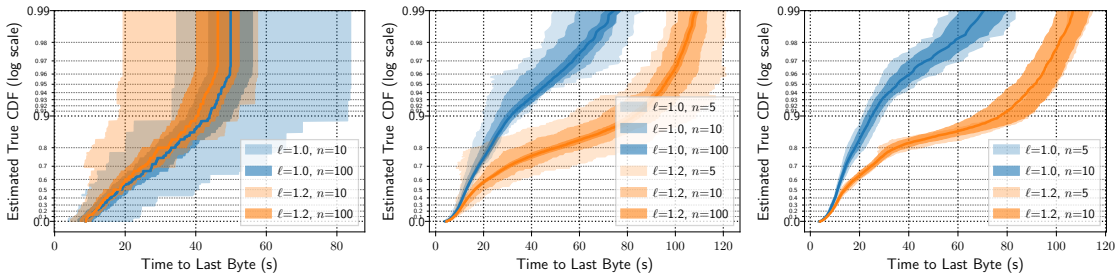
(a) 1% Network Scale ($s = 0.01$) (b) 10% Network Scale ($s = 0.1$) (c) 30% Network Scale ($s = 0.3$)

Figure A.1: Time to first byte in seconds of all 50 KiB, 1 MiB, and 5 MiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s . The results from each experiment are aggregated from n simulations following §3.5, and the CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance.



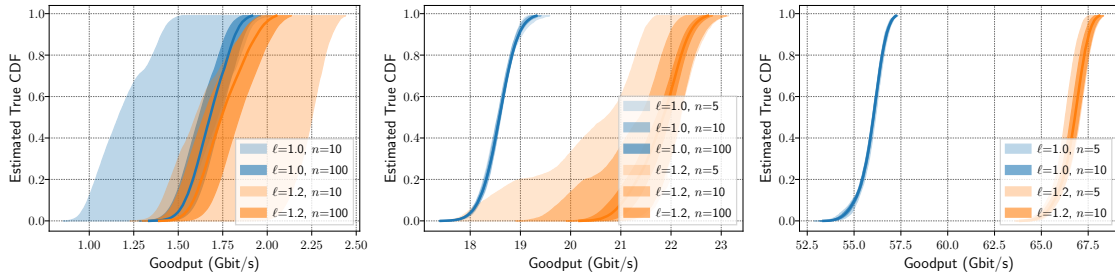
(a) 1% Network Scale ($s = 0.01$) (b) 10% Network Scale ($s = 0.1$) (c) 30% Network Scale ($s = 0.3$)

Figure A.2: Time to last byte in seconds of 50 KiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s . The results from each experiment are aggregated from n simulations following §3.5, and the CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance.



(a) 1% Network Scale ($s = 0.01$) (b) 10% Network Scale ($s = 0.1$) (c) 30% Network Scale ($s = 0.3$)

Figure A.3: Time to last byte in seconds of 5 MiB downloads from performance benchmarking clients from experiments with traffic load $\ell = 1.0$ and $\ell = 1.2$ in networks of various scale s . The results from each experiment are aggregated from n simulations following §3.5, and the CDFs are plotted with tail-logarithmic y-axes in order to highlight the long tail of network performance.



(a) 1% Network Scale ($s = 0.01$) (b) 10% Network Scale ($s = 0.1$) (c) 30% Network Scale ($s = 0.3$)
Figure A.4: Tor network goodput in Gbit/s: for each second in the simulation, the sum of the goodput from all Tor relays. The results from each experiment are aggregated from n simulations following §3.5.

Appendix B

Rust Vulnerabilities in Oxidation

This is a list of descriptions of all the vulnerabilities we identified in Rust projects from relevant Oxidation components (i.e., projects primarily consisting of Rust code that is, was, or was planned to be shipped in Firefox). For an overview and more holistic analysis, see Section 5.3.1. All vulnerabilities are identified by their Bugzilla IDs.

1420001: A history leak. Visited links have a different style applied than unvisited links (notably, the color changes). In WebRender, this was true for links in SVG images as well. A malicious page could query whether a user had visited any particular URL by generating an SVG file with a link to that URL in it, saving the SVG as another image format (e.g., PNG), and checking the color of the pixels where the text is located. The original C++ CSS styling had the same bug, but fixed it, with WebRender causing it to be reintroduced. There is no associated VCC because Servo has had visited link styling disabled from its inception through to the time of this writing for exactly this reason—it was the use of Firefox’s existing style rules on Stylo that caused the issue to arise.

1557208: A potential heap use-after-free on browser shutdown in WebRender, caused by a race condition where C++ code could free memory from shared threads still in use by Rust code. Mozilla developers believe it was “[not] realistically exploitable”, but tracked it as a vulnerability to be safe.

1577439: A potential heap use-after-free on browser shutdown in Stylo, caused by a race condition where C++ code could free memory from shared threads still in use by Rust code. Mozilla developers believe it was “not exploitable”, but tracked it as a vulnerability to be safe.

- 1593865:** A potential heap use-after-free on browser shutdown in Stylo, caused by a race condition where C++ code could free memory from shared threads still in use by Rust code. Mozilla developers believe it was “[not] extremely dangerous”, but tracked it as a vulnerability to be safe.
- 1599181:** CSS pasted from the user’s clipboard was insufficiently sanitized by Stylo.
- 1602843:** CSS pasted from the user’s clipboard was insufficiently sanitized by Stylo.
- 1614971:** A heap use-after-free in cubeb-coreaudio-rs, caused by a race condition with certain C++ callback threads. The original C++ version in cubeb_audiounit had the same vulnerability.
- 1622291:** A heap use-after-free in cubeb-coreaudio-rs, caused by a race condition with certain C++ callback threads. The original C++ version in cubeb_audiounit had a distinct but similar vulnerability (not counted in the C++ data as it was never fixed).
- 1631232:** A bug in Stylo’s garbage collector for CSS selector rules could lead to an inconsistent internal state, allowing for various forms of memory corruption.
- 1637112:** A boundary error in a check (i.e., a > should have been >=) allowed WebRender to render 0-width or 0-height images as arbitrary graphics memory to the screen. Web pages do not have access to the rendered page in normal circumstances, but if combined with some other theoretical vulnerability that broke this boundary, it would have allowed reading graphics memory from any process.
- 1668514:** A soundness bug in Crossbeam. The bug was a result of a faulty assumption in the underlying layout model of Rust’s vector implementation. It was never determined if Firefox had any code that could trigger this bug in practice.
- 1680084:** CSS pasted from the user’s clipboard was improperly sanitized by Stylo.
- 1685145:** A soundness bug where an unsafe block inside a function not annotated as unsafe in WebRender did not check if a pointer was non-null before measuring the size of its allocation, allowing a web page to possibly crash the process with a null pointer exception.
- 1694670:** A bounds check was added to qcms that, when triggered, was inverted, and so only allowed invalid color profiles on Linux. The bounds check was intended to prevent out-of-bounds writes in unsafe code, so the issue was tracked as a security vulnerability. However, the check did not exist at all prior to the introduction of the bug, and the out-of-bounds write could only be triggered by a malicious window server (i.e., part of the operating system), so was extremely unlikely to present any actual security risk.

- 1696312:** A raw pointer in a job queue in WebRender was being cached and not cleared when the queue was empty, allowing for a use-after-free to occur. Unlike similar bugs, this unsafe pointer management was being done for performance reasons, not for language-boundary compatibility.
- 1700235:** As an optimization, WebRender added support for cropping each layer as it was built, rather than applying after all layers were composed. A page that then scaled the result of these operations could amplify floating point rounding errors, allowing it to render web content over the interface of the browser itself (e.g., changing the apparent text of the URL of the current page).
- 1701834:** Another WebRender bug that allowed rendering over the browser interface. The flaw itself has to do with the details of 3D transformation code. The VCC changed the behavior of a function designed to make the resulting x and y values of a given 3D transformation unaffected by z values, “flattening” the transformation; the VCC made the transformation matrix also leave the resulting z value unaffected by the initial x and y values, causing unanticipated behavior when composed with non-flattened transforms.
- 1716028:** Similar to the other upstream Crossbeam bug (1668514), a soundness bug originating from a faulty assumption in the capacity of Rust vectors created from iterators.
- 1746545:** The VCC added the `-ffast-math` compiler flag to part of WebRender’s build script (written in Rust). This allowed the compiler to assume floating point numbers would never be NaN or Inf, in turn optimizing out WebRender’s existing safety checks for handling edge cases, like division by zero.
- 1758223:** A race condition on calls from Stylo to a C++ function to set/unset flags. Was deemed unlikely to be exploitable.

Glossary

chardet The original character encoding detection library written in C++ for Firefox. 78

chardetng A character encoding detection library written in Rust. 78

circuit A path through the Tor network shared by some associated set of streams. 5–8, 17, 21–23, 27, 44, 50, 51, 55, 58, 61, 65–67, 70, 125

Crossbeam A popular Rust library providing concurrent code utilities. It is used in several Mozilla Rust projects, but is not developed by them. 86, 87, 122, 123

cubeb-coreaudio-rs A Rust library for interfacing with the MacOS audio system. 78, 122

cubeb_audiounit The original Firefox C++ library for interfacing with the MacOS audio system. 78, 122

directory authority One of the small number of Tor relays that collectively construct and publish the network consensus. 6, 39, 125

dyadic Relating to two—here, meaning conversations with exactly two participants. 53, 55, 61, 63, 64

e2ee End-to-end encryption or end-to-end encrypted—i.e., contents are encrypted such that only the endpoints can access them. 1–4, 45–48, 71, 73, 95, 96

encoding_rs A Rust library for encoding Unicode strings. 78

exit relay A Tor relay configured to allow establishing connections that leave the Tor network. 6, 8, 19–22, 39, 55–57, 60, 69, 125

Gecko The browser engine serving the core of Mozilla products—most notably Firefox, but also sibling projects such as the Thunderbird email client. 78, 79, 85, 126

guard relay A Tor relay serving as the first hop in a circuit. These relays are selected for their stability, and rarely change once selected by a Tor client, even when constructing new circuits. 5–7, 19–22, 31, 39, 44, 125

Layers The original rendering engine written in C++ for Firefox. 78

libhyphen A C++ library for choosing where in wrapped text to insert hyphens. It was originally written as part of the Hunspell project, but was vendored for use in Firefox. 78, 86

mapped_hyph A Rust library for choosing where in wrapped text to insert hyphens. 78

middle relay A Tor relay that is not operating as a guard relay or exit relay. 6, 7, 19, 21, 39, 69

mp4parse-rust A Rust library for parsing MP4 video files. 78

network consensus The consensus on what Tor relays are currently operating, how to connect to them, their respective bandwidth weights, and what they can be used for, constructed and published hourly in a document by the directory authorities. xii, 6, 14, 17, 19–22, 25, 44, 56, 124, 125

onion service A service that uses Tor to provide anonymous, authenticated, and encrypted routing to it. Formerly known as a “hidden service”. xii, 7–9, 16, 46, 50, 51, 58–61, 67–71, 95

PET Privacy Enhancing Technology—i.e., a technology that provides additional privacy to its user. 1–4, 73, 74, 95, 96

qcms The Firefox library for managing color profiles. Refers to both the original C++ implementation, and the Rust rewrite. 78, 122

server descriptor Additional information a client can query about a particular relay, but that does not need to be directly provided in the frequently updated main network consensus file—e.g., the relay’s public key. 6, 20, 44

Servo An experimental web browser written in Rust. Originally developed by Mozilla, it is now an independent project under the stewardship of the Linux Foundation Europe umbrella organization. 77–79, 85, 121, 126

stagefright A C++ library for parsing various forms of media, originally developed as part of Android, but then vendored into Firefox for MP4 video file parsing. 78, 85, 86

stream A transport-layer connection between two network endpoints—a TCP stream, most typically. 5, 7, 17, 22, 23, 39, 54, 55, 65, 124

style The original CSS style system for Gecko/Firefox, written in C++. The name is somewhat implicit, likely referring to the folder in the source tree it was developed in, and is sometimes referred by descriptive terms like “the Gecko style system” or “the old CSS engine” instead. 78

Stylo A CSS style system written in Rust for Servo, and now also used by Firefox. Sometimes also called “Quantum CSS”. 78, 79, 85, 87, 121–123

SZZ The most popular automated technique for identifying the commit(s) that introduced a bug given the commit(s) that fixed it. Named for the authors’ initials in the work that introduced it [SZZ05]. 75–77, 80–82, 88, 89

uconv A C++ library for encoding Unicode strings. 78

VCC Vulnerability Contributing Commit—i.e., a commit identified as introducing a vulnerability. 75, 76, 80–82, 85, 86, 88, 89, 91, 121, 123

vendor To take some source code of another project (typically a library) and copy it directly into this project’s source tree, rather than linking to an externally maintained repository or using a dependency management system. 78, 86, 125, 126

WebRender A rendering engine originally written in Rust for Servo, but then vendored and maintained in Firefox. Sometimes also called “Quantum Render”. 78, 85, 87, 121–123