6 Appendix

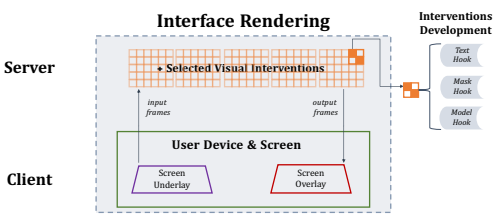
6.1 GreaseTerminator

In response to the continued widespread presence of interface-based harms in digital systems, Datta et al. (Datta et al., 2021) developed *GreaseTerminator*, a visual overlay modification method. This approach enables researchers to develop, deploy and study interventions against interface-based harms in apps. This is based on the observation that it used to be difficult in the past for researchers to study the efficacy of different intervention designs against harms within mobile apps (most previous approaches focused on desktop browsers). *GreaseTerminator* provides a set of 'hooks' that serve as templates for researchers to develop interventions, which are then deployed and tested with study participants. *GreaseTerminator* interventions usually come in the form of machine learning models that build on the provided hooks, automatically detect harms within the smartphone user interface at run-time, and choose appropriate interventions (e.g. a visual overlay to hide harmful content, or content warnings). The *GreaseTerminator* architecture is shown in Figure 6(a) in contrast to the *GreaseVision* architecture.

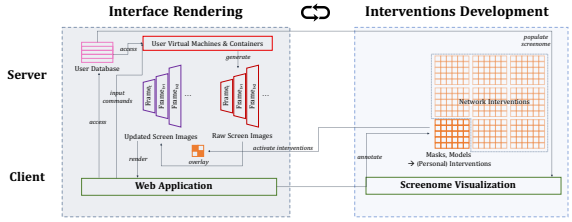
Technical improvements w.r.t. *GreaseTerminator* The improvements of *GreaseVision* with respect to *GreaseTerminator* are two-fold: (i) improvements to the framework enabling end-user development and harms mitigation (discussed in detail in Sections 4.2, 4.3, 5 and 6), and (ii) improvements to the technical architecture (which we discuss in this section). Our distinctive and non-trivial technical improvements to the *GreaseTerminator* architecture fall under namely latency, device support, and interface-agnosticity. *GreaseTerminator* requires the end-user device to be the host device, and overlays graphics on top. A downside of this is the non-uniformity of network latency between users (e.g. depending on the internet speed in their location) resulting in a potential mismatch in rendered overlays and underlying interface. With *GreaseVision*, we send a post-processed/re-rendered image once to the end-user device's browser (stream buffering) and do not need to send any screen image from the host user device to a server, thus there is no risk of overlay-underlay mismatch and we even reduce network latency by half. Images are relayed through an HTTPS connection, with a download/upload speed $\sim$ 250Mbps, and each image sent by the server amounting to $\sim$ 1Mb). The theoretical latency per one-way transmission should be $\frac{1\times1024\times8\text{bits}}{250\times10^6\text{bits/s}}$ = 0.033ms. With each user at most requiring server usage of one NVIDIA GeForce RTX 2080, with reference to existing online benchmarks (Ignatov, 2021) the latency for 1 image (CNN) and text (LSTM) model would be 5.1ms and 4.8ms respectively. While the total theoretical latency for *GreaseTerminator* is $(2 \times 0.033 + 5)$, that of *GreaseVision* is $(0.033 + 5)$ = 5.03ms. Another downside of *GreaseTerminator* is that it requires client-side software for each target platform. There would be pre-requisite OS requirements for the end-user device, where only versions of *GreaseTerminator* developed for each OS can be offered support (currently only for Android). *GreaseVision* streams screen images directly to a login-verified browser, allowing users to access desktop/mobile on any browser-supported device. Despite variations in the streaming architecture between *GreaseVision* and *GreaseTerminator*, the interface modification framework (hooks and overlays) are retained, hence interventions (even those developed by end-users) from *GreaseVision* are compatible in *GreaseTerminator*. In addition to improvements to the streaming architecture to fulfil interface-agnosticity, adapting the visual overlay modification framework into a collaborative HITL implementation further improves the ease-of-use for all stakeholders in the ecosystem. End-users do not need to root their devices, find intervention tools or even self-develop their own customized tools. We eliminate the need for researchers to craft interventions (as users self-develop autonomously) or develop their own custom experience sampling tools (as end-users/researchers can analyze digital experiences from stored screenomes). We also eliminate the need for intervention developers to learn a new technical framework or learn how to fine-tune models. Running emulators on docker containers and virtual machines on a (single) host server is feasible, and thus allows for the browser stream to be accessible cross-device without restriction, e.g. access iOS emulator on Android device, or macOS virtual machine on Windows device. Certain limitations are imposed on the current implementation, such as a lack of access to the device camera, audio, and haptics; however, these are not permanent issues, and engineered implementations exist where a virtual/emulated device can route and access the host device's input/output sources (VrtualApp, 2016).

Proceedings of the Sixth Workshop on Online Abuse and Harms (WOAH), pages 24 - 28

Figure 6: Architecture of *GreaseTerminator* (left) and *GreaseVision* (right).



(a) The high-level architecture of *GreaseTerminator*. Details are explained in Section 2.3 and 4.2.



(b) The high-level architecture of *GreaseVision*, both as a summary of our technical infrastructure as well as one of the collaborative HITL interventions development approach.

Hooks The *text hook* enables modifying the text that is displayed on the user's device. It is implemented through character-level optical character recognition (OCR) that takes the screen image as an input and returns a set of characters and their corresponding coordinates. The EAST text detection (Zhou et al., 2017) model detects text in images and returns a set of regions with text, then uses Tesseract (Google, 2007) to extract characters within each region containing text. The *mask hook* matches the screen image against a target template of multiple images. It is implemented with *multi-scale multi-template* matching by resizing an image multiple times and sampling different subimages to compare against each instance of mask in a masks directory (where each mask is a cropped screenshot of an interface element). We retain the default majority-pixel inpainting method for mask hooks (inpainting with the most common colour value in a screen image or target masked region). As many mobile interfaces are standardized or uniform from a design perspective compared to images from the natural world, this may work in many instances. The mask hook could be connected to rendering functions such as highlighting the interface element with warning labels, or image inpainting (fill in the removed element pixels with newly generated pixels from the background), or adding content/information (from other apps) into the inpainted region. Developers can also tweak how the mask hook is applied, for example using the multi-scale multi-template matching algorithm with contourized images (shapes, colour-independent) or coloured images depending on whether the mask contains (dynamic) sub-elements, or using few-shot deep learning models if similar interface elements are non-uniform. A *model hook* loads any machine learning model to take any input and generate any output. This allows for model embedding (i.e. model weights and architectures) to inform further overlay rendering. We can connect models trained on specific tasks (e.g. person pose detection, emotion/sentiment analysis) to return output given the screen image (e.g. bounding box coordinates to filter), and this output can then be passed to a pre-defined rendering function (e.g. draw filtering box).

6.2 Related Works (extended)

6.2.1 Motivation: Pervasiveness and Individuality of Digital Harms

It is well-known that digital harms are widespread in our day-to-day technologies. Despite this, the academic literature around these harms is still developing, and it remains difficult to state exactly what the harms are that need to be addressed. Famously, Gray et al. (Gray et al., 2018) put forward a 5-class taxonomy to classify dark patterns within apps: *interface interference* (elements that manipulate the user interface to induce certain actions over other actions), *nagging* (elements that interrupt the user's current task with out-of-focus tasks) *forced action* (elements that introduce sub-tasks forcefully before permitting a user to complete their desired task), *obstruction* (elements that introduce subtasks with the intention of dissuading a user from performing an operation in the desired mode), and *sneaking* (elements that conceal or delay information relevant to the user in performing a task).

A challenge with such framework and taxonomies is to capture and understand the material impacts of harms on individuals. Harms tend to be highly individual and vary in terms of how they manifest within users of digital systems. The harms landscape is also quickly changing with ever-changing digital systems. Defining the spec-

trum of harms is still an open problem, the range varying from heavily-biased content (e.g. disinformation, hate speech), self-harm (e.g. eating disorders, self-cutting, suicide), cyber crime (e.g. cyber-bullying, harassment, promotion of and recruitment for extreme causes (e.g. terrorist organizations), to demographic-specific exploitation (e.g. child-inappropriate content, social engineering attacks) (HM, 2019; Pater and Mynatt, 2017; Wang et al., 2017; Honary et al., 2020; Pater et al., 2019), for which we recommend the aforementioned cited literature. The last line of defense against many digital harms is the user interface. This is why we are interested in interface-emergent harms in this paper, and how to support individuals in developing their own strategies to cope with and overcome such harms.

6.2.2 Developments in Interface Modification & Re-rendering

Digital harms have long been acknowledged as a general problem, and a range of technical interventions against digital harms are developed. Interventions, also similarly called modifications or patches, are changes to the software, which result in a change in (perceived) functionality and end-user usage. We review and categorize key *technical* intervention methods for interface modification by end-users, with cited examples specifically for digital harms mitigation. While there also exist non-technical interventions, in particular legal remedies, it is beyond this work to give a full account of these different interventions against harms; a useful framework for such an analysis is provided by Lawrence Lessig (Lessig) who characterised the different regulatory forces in the digital ecosystem.

Interface-code modifications (Kollnig et al., 2021; Higi, 2020; Jeon et al., 2012; Rasthofer et al., 2014; Davis and Chen, 2013; Backes et al., 2014; Xu et al., 2012; LuckyPatcher, 2020; Davis et al., 2012; Lyngs et al., 2020b; Freeman, 2020; rovo89, 2020; Agarwal and Hall, 2013; Enck et al., 2010; MaaarZ, 2019; VrtualApp, 2016) make changes to source code, either installation code (to modify software before installation), or run-time code (to modify software during usage). On desktop, this is done through *browser extensions* and has given rise to a large ecosystem of such extensions. Some of the most well-known interventions are ad blockers, and tools that improve productivity online (e.g. by removing the Facebook newsfeed (Lyngs et al., 2020b)). On mobile, a prominent example is *AppGuard* (Backes et al., 2014), a research project by Backes et al. that allowed users to improve the privacy properties of apps on their phone by making small, targeted modification to apps' source code. Another popular mobile solution in the community is the app *Lucky Patcher* (LuckyPatcher, 2020) that allows to get paid apps for free, by removing the code relating to payment functionality directly from the app code.

Some of these methods may require the highest level of privilege escalation to make modifications to the operating system and other programs/apps as a root user. On iOS, *Cydia Substrate* (Freeman, 2020) is the foundation for jailbreaking and further device modification. A similar system, called *Xposed Framework* (rovo89, 2020), exists for Android. To alleviate the risks and challenges afflicted with privilege escalation, *VirtualXposed* (VrtualApp, 2016) create a virtual environment on the user's Android device with simulated privilege escalation. Users can install apps into this virtual environment and apply tools of other modification approaches that may require root access. *ProtectMyPrivacy* (Agarwal and Hall, 2013) for iOS and *TaintDroid* (Enck et al., 2010) for Android both extend the functionality of the smartphone operating system with new functionality for the analysis of apps' privacy features. On desktops, code modifications tend not to be centred around a common framework, but are more commonplace in general due to the traditionally more permissive security model compared to mobile. Antivirus tools, copyright protections of games and the modding of UI components are all often implemented through interface-code modifications.

Interface-external modifications (Geza, 2019; Bodyguard, 2019; Lee et al., 2014; Ko et al., 2015; Andone et al., 2016; Hiniker et al., 2016; Löchtefeld et al., 2013; Labs, 2019; Okeke et al., 2018) are the arguably most common way to change default interface behaviour. An end-user would install a program so as to affect other programs/apps. No change to the operating system or the targeted programs/apps is made, so an uninstall of the program providing the modification would revert the device to the original state. This approach is widely used to track duration of device usage, send notifications to the user during usage (e.g. timers, warnings), block certain actions on the user device, and other aspects. The *HabitLab* (Geza, 2019) is a prominent example developed by Kovacs et al. at Stanford.

This modification framework is open-source and maintained by a community of developers, and provides interventions for both desktop and mobile.

Visual overlay modifications render graphics on an overlay layer over any active interface instance, including browsers, apps/programs, videos, or any other interface in the operating system. The modifications are visual, and do not change the functionality of the target interface. It may render sub-interfaces, labels, or other graphics on top of the foreground app. Prominent examples are *DetoxDroid* (flxapps, 2021), *Gray-Switch* (GmbH, 2021), *Google Accessibility Suite* (Google, 2021), and *GreaseTerminator* (Datta et al., 2021).

We would like to establish early on that we pursue a *visual overlay modifications* approach. Interventions should be rendered in the form of overlay graphics based on detected elements, rather than implementing program code changes natively, hence focused on changing the interface rather than the functionality of the software. Interventions should be generalizable; they are not solely website- or app-oriented, but *interface-oriented*. Interventions do not target specific apps, but general interface elements and patterns that could appear across different interface environments. To support the systemic requirements in Section 2.4, we require an interface modification approach that is (i) interface-agnostic and (ii) easy-to-use. To this extent, we build upon the work of ***GreaseTerminator*** (Datta et al., 2021), a framework optimized for these two requirements.

In response to the continued widespread presence of interface-based harms in digital systems, Datta et al. (Datta et al., 2021) developed *GreaseTerminator*, a visual overlay modification method. This approach enables researchers to develop, deploy and study interventions against interface-based harms in apps. This is based on the observation that it used to be difficult in the past for researchers to study the efficacy of different intervention designs against harms within mobile apps (most previous approaches focused on desktop browsers). *GreaseTerminator* provides a set of 'hooks' that serve as templates for researchers to develop interventions, which are then deployed and tested with study participants. *GreaseTerminator* interventions usually come in the form of machine learning models that build on the provided hooks, automatically detect harms within the smartphone user interface at run-time, and choose appropriate interventions (e.g. a visual overlay to hide harmful content, or content warnings). A visualisation of the *GreaseTerminator* approach is shown in Figure 6(a).

6.2.3 Opportunities for Low-code Development in Interface Modification

Low-code development platforms have been defined, according to practitioners, to be (i) low-code (negligible programming skill required to reach endgoal, potentially drag-and-drop), (ii) visual programming (a visual approach to development, mostly reliant on a GUI, and "what-you-see-is-what-you-get"), and (iii) automated (unattended operations exist to minimize human involvement) (Luo et al., 2021). Low-code development platforms exist for varying stages of software creation, from frontend (e.g. App maker, Bubble.io, Webflow), to workflow (Airtable, Amazon Honeycode, Google Tables, UiPath, Zapier), to backend (e.g. Firevase, WordPress, flutterflow); none exist for software modification of existing applications across interfaces. According to a review of StackOverflow and Reddit posts analysed by Luo et al. (Luo et al., 2021), low-code development platforms are cited by practitioners to be tools that enable faster development, lower the barrier to usage by non-technical people, improves IT governance compared to traditional programming, and even suits team development; one of the main limitations cited is that the complexity of the software created is constrained by the options offered by the platform.

User studies have shown that users can self-identify malevolent harms and habits upon self-reflection and develop desires to intervene against them (Cho et al., 2021; Lyngs et al., 2020a). Not only do end-users have a desire or interest in self-reflection, but there is indication that end-users have a willingness to act. Statistics for content violation reporting from Meta show that in the Jan-Jun 2021 period, $\sim$ 42,200 and $\sim$ 5,300 in-app content violations were reported on Facebook and Instagram respectively (Meta, 2022) (in this report, the numbers are specific to violations in local law, so the actual number with respect to community standard violatons would be much higher; the numbers also include reporting by governments/courts and non-government entities in addition to members of the public). Despite a willingness to act, there are limited digital visualization or reflection tools that enable flexible intervention development

by end-users. There are visualization or reflection tools on browser and mobile that allow for reflection (e.g. device use time (Andone et al., 2016)), and there are separate and disconnected tools for intervention (Section 2.2), but there are limited offerings of flexible intervention development by end-users, where end-users can observe and analyze their problems while generating corresponding fixes, which thus prematurely ends the loop for action upon regret/reflection. There is a disconnect between the harms analysis ecosystem and interventions ecosystem. A barrier to binding these two ecosystems is the existence of low-code development platforms for end-users. While such tooling may exist for specific use cases on specific interfaces (e.g. web/app/game development) for mostly creationary purposes, there are limited options available for modification purposes of existing software, the closest alternative being extension ecosystems (Kollnig et al., 2021; Google, 2010a). Low-code development platforms are in essence "developer-less", removing developers from the software modification pipeline by reducing the barrier to modification through the use of GUI-based features and negligible coding, such that end-users can self-develop without expert knowledge.

Human-in-the-Loop (HITL) learning is the procedure of integrating human knowledge and experience in the augmentation of machine learning models. It is commonly used to generate new data from humans or annotate existing data by humans. Wallace et al. (Wallace et al., 2019) constructed a HITL system of an interactive interface where a human talks with a machine to generate more Q&A language and train/fine-tune Q&A models. Zhang et al. (Zhang et al., 2019) proposed a HITL system for humans to provide data for entity extraction, including requiring humans to formulate regular expressions and highlight text documents, and annotate and label data. For an extended literature review, we refer the reader to Wu et al. (Wu et al., 2021). Beyond lab settings, HITL has proven itself in wide deployment, where a wide distribution of users have indicated a willingness and ability to perform tasks on a HITL annotation tool, *reCAPTCHA*, to access utility and services. In 2010, Google reported over 100 million reCAPTCHA instances are displayed every day (Google, 2010b) to annotate different types of data, such as deciphering text for OCR of books or street signs, or labelling objects in images such as traffic lights or vehicles.

While HITL formulates the structure for human-AI collaborative model development, **model fine-tuning** and **few-shot learning** formulate the algorithmic methods of adapting models to changing inputs, environments, and contexts. Both adaptation approaches require the model to update its parameters with respect to the new input distribution. For model fine-tuning, the developer re-trains a pre-trained model on a new dataset. This is in contrast to training a model from a random initialization. Model fine-tuning techniques for pre-trained foundation models, that already contain many of the pre-requisite subnetworks required for feature reuse and warm-started training on a smaller target dataset, have indicated robustness on downstream tasks (Galanti et al., 2022; Abnar et al., 2022; Neyshabur et al., 2020). If there is an extremely large number of input distributions and few samples per distribution (small datasets), few-shot learning is an approach where the developer has separately trained a meta-model that learns how to change model parameters with respect to only a few samples. Few-shot learning has demonstrated successful test-time adaptation in updating model parameters with respect to limited test-time samples in both image and text domains (Raghu et al., 2020; Koch et al., 2015; Finn et al., 2017; Datta, 2021). Some overlapping techniques even exist between few-shot learning and fine-tuning, such as constructing subspaces and optimizing with respect to intrinsic dimensions (Aghajanyan et al., 2021; Datta and Shadbolt, 2022; Simon et al., 2020).

The raw data for harms and required interface changes reside in the history of interactions between the user and the interface. In the Screenome project (Reeves et al., 2020, 2021), the investigators proposed the study and analysis of the moment-by-moment changes on a person's screen, by capturing screenshots automatically and unobtrusively every $t = 5$ seconds while a device is on. This record of a user's digital experiences represented as a sequence of screens that they view and interact with over time is denoted as a user's **screenome**. Though not mobilized widely amongst users for their self-reflection or personalized analysis, integrating screenomes into an interface modification framework can play the dual roles of visualizing raw (harms) data to users while manifesting as parseable input for visual overlay modification frameworks.