Using Context to Verify Human Intent

by

He (Shawn) Shuang

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Using Context to Verify Human Intent

He (Shawn) Shuang
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2020

We show that all defenses [39, 51, 59, 66] for operating system-level user impersonation attack (OS-UImp) fall prey to a new class of user interface(UI) attacks — Context Forgery (CF). It can be executed by the same attacker as OS-UImp and cause the same consequence: unintended requests are accepted and executed by services.

We design and build AINT to protect users from deceptive UI and capture human intent. AINT captures a video recording of the display which contains the rendering and user interactions. The rendering is validated against specifications from the service to ensure UI integrity. To extract human intent, AINT relies on a non-malicious user to validates her inputs is displayed properly on the screen — a process we call Implicit Confirmation (IC). This allows AINT to extract user inputs from the video recording and use that as the human intent.

Acknowledgements

I would like to thank my supervisor, Professor David Lie, for his support and guidance throughout the course of study of my Master's degree. I greatly appreciate the time and care he spends on each student.

I would also like to thank Lianying(Viao) Zhao and Wei Huang for their help and discussions.

I would like to thank Professor Baochun Li for his support.

I am also grateful for the financial support from the University of Toronto and Ontario government for the scholarships.

Lastly, I would like to thank my family and friends for supporting me as I complete my degree. I would like to specially thank Piaoyao Shi for her support throughout my study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A malware on the client [1] may generate service requests to a remote service that is unintended by the human user. For instance, a malware on Alice's client may attempt to construct and send a transfer request to Paypal without Alice's awareness. We call this type of attack a user impersonation attack.

One possible defense is CAPTCHA. CAPTCHA uses problems easy for human, but difficult for machines (and thus malware), to infer whether a request is human-generated. However, CAPTCHA falls prey to a more powerful attacker with an operating system(OS)-level privilege. An OS-level attacker may tamper with program memory stealthily so that the service request generated is not human intended. We call this type of attack an OS-level user impersonation attack (OS-UImp).

The root cause of OS-UImp is that the human intent is not directly exposed to the service. The state-of-the-art defenses [66, 39, 59] for OS-UImp all require the user to perceive the UI and perform extra work correctly such as identifying a trusted display and compare two set of text. We challenge this assumption. Prior work on UI attacks [78, 90, 54, 87] has shown the feasibility to affect user perception and trick users into carrying out unintended actions. We identify a new class of UI attack called Context Forgery (CF), where an OS-level attacker attacks system with application memory protection. CF tries to trick the user into carrying out unintended actions through modifying, resizing, overlaying or covering UI elements. The focus of CF is to lure the user to construct unintended service requests — the same goal as OS-UImp; CF can be carried out the same attacker as OS-UImp — an OS-level attacker. All of the existing defense for OS-UImp [51, 59, 39, 66] fall prey to CF.

In summary, the limitations of the state-of-the-art defenses of OS-UImp are:

- **Extra user effort** is required.
- Vulnerable to **Context Forgery** attacks.

Thesis Statement. We believe it is possible to build a solution to OS-UImp while avoiding the two limitations. We hypothesize that it is possible to build a solution that "sees what the user sees". Specifically, to prevent context forgery attacks on a web page, a solution is to ensure that what user sees matches the expected appearance of the web page designer, preventing client-side tampering. And to prevent user impersonation attack, by "seeing" how the user interacts with the page, we can infer her intention. By passively "seeing what the user sees", no extra user effort is required. We illustrate our solution in the rest of this section.

[1] In this thesis, we use the term *client* to denote any computing device, including smartphones, PCs and even embedded/IoT devices.

We propose Attested Intention(AINT), a framework that defeats both OS-UImp and CF without trusting the OS or any applications. The main idea of AINT is to record user display and conclude 1) whether what the user sees is correct and 2) extract the human user's intent based on how the user interaction on user interface. To achieve our objectives, AINT extracts the display and validates it against specifications from services. Without rely on the OS, AINT proposes a design guideline, called tabularization, for remote services to design user interfaces that is verifiable by an external observer. Tabularization requires the service to place UI elements into cells so that the rendering of cells can be individually validated, which divides the difficult problem of validating the entire page into simpler problems of validating cells. AINT uses two metrics to validate the rendering: image hash and optical character recognition (OCR). Image hashes hash visually similar images to similar values. It provides robustness to rendering variations. OCR extracts text from images and provides detection ability on subtle changes such as single character change. A set of *good* hashes and text are contained inside the network packets pre-calculated by the service and are used to validate client-side rendering. A validated UI ensures that user will not be tricked by illegal UI modifications, and future ensures that the user inputs through peripherals follows her intent.

To capture semantic-rich human intent, AINT relies on an important property: a non-malicious user always ensures the display of her inputs matches with her inputs through peripherals, a process we call *Implicit Confirmation(IC)*. For instance, as the user types 'abc' on the keyboard, she ensures the display shows 'abc' at the place she expects. With IC, AINT can extract human intention solely from the display — the textual inputs represents human intent. However, we believe it is infeasible for the human user to validate every character on the screen all the time, thus, AINT developed two techniques to reduce user effort. First of all, AINT tracks user focus and only extracts text currently under the user focus. Once, the user has finished interacting with a field, she no longer needs to validate it. Secondly, the blinking input cursor suggests the position of user focus in a long paragraph, thus, AINT only takes into account the most recent characters the user occurred on the right-side of the cursor. The user does not have to validate previously entered characters. AINT enforces these two rules using a combination of computer vision and optical character recognition (OCR). To expose the human intent to the service, AINT generates the service requests with raw human intent in a Trusted Execution Environment (TEE). AINT requires the service to only accept requests generated from AINT, a malicious OS can generate its request, but it will not be accepted by the remote service.

Since AINT requires server-side change, we simulate the server-side work on five web pages. On the client-side, we implemented AINT using Xen, Tesseract, and OpenCV. We also added a cache in our implementation to speed up AINT's performance. We evaluate our method of validating the client-side rendering using various UI attacks we simulated. We show AINT the ability to do validation and where it fails in certain circumstances. We also evaluate the performance of AINT in a CPU-only setting, as well as with Tesseract running in a GPU. Since AINT relies on and is the first to use, image hashes for tampering detection, and image hash provides no security guarantees comparing to a cryptographic hash function, we evaluate the similarity tolerance, collision rate and the performance impact of various image hash functions. The result shows that our choice of image hash needs improvements.

This thesis focuses on the design and implementation of a prototype that deals with web interface on a traditional x86 environment, where users interact with computers through mice and keyboards. We believe that the methodology is applicable to the user interface on the mobile platform such as Android, where the main interaction device is touch screens.

Contributions. Our contributions include the following:

- We identify Context Forgery(CF), a class of user interface(UI) attack that attempts to trick the user into carrying out unintended actions. Both CF and OS-level user impersonation attacks (OS-UImp) cause user unintended service requests to be sent to remote services. We show how current defenses of OS-UImp are vulnerable CF.
- We propose the design of AINT, a framework that prevents both OS-UImp and CF. AINT guarantees that users will not be tricked by an OS-level attacker and guarantees that the user's intent is properly delivered to the remote service.
- We implement a prototype of AINT on the x86 platform.
- We present an evaluation of AINT, including its security guarantees and performance.

1.1 Thesis Organization

The rest of this thesis is organized as follows: we first list some of the related work in Chapter 2. We introduce Context Forgery(CF) attack in Chapter 3. We then explore the design space of our defense AINT in Chapter 4, followed by its implementation in Chapter 5 and evaluation in Chapter 6. We list the future work in Chapter 7. Finally, we conclude in Chapter 8.

Chapter 2

Related Work

This chapter discusses related work. We group the related works into three categories: 1) UI attacks and defenses 2) protect program execution: protect the software in a malicious execution environment and 3) account for the physical human in digital computers: bridging the semantic gap between a physical human and a computer.

2.1 User Interface(UI) Attacks

We define user interface (UI) attacks as attacks that aim to affect user perception through the user interface of digital devices. UI attacks broadly include phishing [58], where the user sees the UI from an impersonating principal, and clickjacking, where the user sees attacker-controlled UI and any user actions from the attacker's UI are used to trigger unintended actions on the legitimate UI. Prior work [14, 71] has shown the effectiveness and root causes of UI attacks, and one cause is the human factor: user may fail to recognize phishing sites [31, 35], fail to recognize subtle tampering [54] and fail to understand the consequences of their activities [88]. Current UI attacks focus on the web interface in x86 and application interface in Android.

2.1.1 Phishing

A well-known form of UI attack is phishing, where an attacker impersonates a legit service. If the human users believes in the deceptive UI and enters any sensitive information, the impersonating principal will get access to that information. The root cause of phishing attacks is UI deception [31]. The defense for phishing, other than user education, have largely focused on the detection of impersonating user interface [62, 75, 1, 117]. AINT does not provide any protection against phishing attacks because AINT requires cooperation from legitimate services.

2.1.2 Clickjacking

The other class of UI attacks is clickjacking attack. As shown in Fig 2.1, the idea is to overlay the UI of one party with attacker-controlled UI, and while the user is interacting with the attacker-controlled UI, her interactions such as clicks and inputs are hijacked and passed to the application underneath [90]. In this case, the user only intends to interact with the application at the front, not the underneath

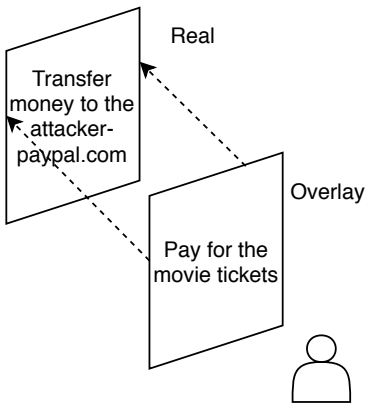


Figure 2.1: Clickjacking explained.

application. Hijacking is not limited to clicks [79], it can be combined with other attacks such as CSRF [60], XSS [78] and CSS [53], to steal files [64] and cookies [111], and with a bit of domain-specific knowledge, it can hijack likes and shares on Facebook [26].

The root cause of this type of attack is again UI deception [2] and how user interaction with one application is illegally delivered to an unintended recipient [44]. Since these two causes are platform-independent, clickjacking has shown effectiveness on both x86 [54] and Android [115].

The defense for UI attack has mainly focused on preventing UI overlay and restricting the delivery of user interaction to unintended recipients. To prevent UI overlay, frame buster [94, 65] is a technique used by web pages to check if they are being contained in a frame, since loading a web page inside a frame is the first step to overlay it on another page. Some web browser such as Gazelle [113] prevents cross-origin frames to be rendered transparently. To stop the delivery of user interaction to other pages, one method is to do double confirmation on security-sensitive operations. For instance, Facebook requires confirmation when the user clicks on the like button [42]. Another set of defense mechanism [48, 10], checks, when an element is being clicked, whether the element is displayed with no overlay. If the element is partially or entirely obscured, when it was clicked, that indicates the possibility of a clickjacking attack.

On the Android side, some work [87, 116] aims to detect malicious overlay through several features such as the color density of the overlay. Android implements a touch filter, a mechanism that acknowledges the application if user IO happens when the view is obscured. Google has adopted this mechanism for important applications such as Setting in Android. However, this can be bypassed by hiding all elements other than the protected button [44], thus creating an entire overlay except the button. As a result, Google entirely prevents overlays on apps such as Setting. This breaks the compatibility of existing wedges. Also, not all Google apps are protected [87]. To prevent user IO from being sent to undesignated apps, one of the early work [79] suggests to prevent user input from passing between apps, obviously, that causes compatibility issues with existing applications.

Our identified CF attack is also a type of UI attack, CF tries to affect user perception using the same techniques used in UI attacks, that includes but not limited to 1) hijacking the user interaction with one application and pass to another application 2) modifying part of the display to change its semantic meaning and 3) crafting the whole screen except certain areas such as any trusted embedded display. Due to the high privilege, none of the clickjacking defense is effective against CF. AINT, our proposed defense, requires web pages to be tabularized. On the client side, AINT validates the rendering of web

pages preventing modifications.

2.2 Protect Program Execution

In this section, we detail works in the area of Isolated Execution Environment(IEE). IEE provides integrity guarantees to protected memory regions.

2.2.1 Pure Software-based IEE

A Correct Kernel. One attempt is to construct an entirely secure and bug-free kernel [37, 95, 9]. The idea is that if the OS is correct and tamper-resistant, it should 1) protect itself from tampering and 2) provide isolation between applications. SeL4 [63] is a microkernel designed for this purpose and it is formally verified against its specifications. However, the problems with secure kernels are 1) performance overhead 2) compatibility issues. AINT aims protect the users on commercially available OSes.

Low-level Isolation Module. Another approach is to leverage certain lower-level software modules. Prior work [72, 123, 20, 52, 68, 19, 18, 30, 46, 100] leverages a trusted hypervisor to provide fine-grained separation between an application and the malicious OS. However, due to the presence of the hypervisor, the trusted computing base(TCB) increases. Large TCB hurts the security guarantees [76].

Verifiable Computing. Verifiable computing [83] does not provide run time security guarantee, but it provides an easy way to verify that the computation was done correctly, thus without tampering. Verifiable computation allows one to establish trust in the integrity of the result, and when combined with homomorphic encryption [105], provides confidentiality of the input and output. However, these solutions are not yet feasible to be used with production systems due to their overhead.

2.2.2 Hardware-based IEE

Trusted execution environment(TEE). Prior work has proposed various definitions of TEE [46, 49, 120, 109], but TEEs are hardware-assisted IEEs designed for memory integrity guarantees: critical parts of a TEE must be in the hardware for relative immutability as well as efficiency.

Intel SGX[74], ARM TrustZone [5, 17] are the two primitive TEEs. These two alone provide integrity guarantees, but prior works [104, 6, 13] have developed higher-level abstractions over these primitives providing better usability. Intel TXT [55] and AMD SVM [4] are hardware features that provide dynamic root of trust(DRTM) [50] for launching new kernel or hypervisors and TPM [103] is a hardware module on the chipset that provides secure storage as well as attestability. Flicker [73] combines the two and implements a TEE. Non-commercial TEEs [16, 41, 82, 27] remain as research prototypes and are not viable.

Non-TEEs. There are hardware-based execution environment on x86 architecture that protects code and data, but are not open to developers. For instance, Intel ME [93] and Intel SMM [57, 8] are both IEEs due to their high privilege in the system. Intel ME is considered with privilege level -3 while Intel SMM is considered at level -2. A malicious OS cannot tamper with the execution of inside them.

Alternative Hardware. This set of work moves the application to different hardware location with physical isolation from the malicious OS. The alternative hardware broadly includes GPU [106], CPU

cache [108], co-processors [98, 61, 7] or even the cloud [24, 28].

Hardware-based IEEs have a smaller TCB, and can achieve better efficiency and are relatively immune to tampering comparing to software-based solutions. However, as more software functionality is stuffed into hardware [12], whether it will hold its advantages remains questionable. AINT needs an IEE to generate the final service request based on extracted human intent. We designed AINT to work with Intel SGX because SGX is the sweet spot between usability and performance overhead, but in our implementation, we run the request inside a trusted virtual machine(VM) due to the choice of our hypervisor.

2.3 Bridging a Physical Human and a Computer

The human intent is a psychological state only existed in the human user's brain, it is difficult for a machine to capture this psychological state. In this section, we detail some works that tries to capture human intent and expose it to another trusted machine.

2.3.1 Capture Human Intent

This set of work details method used to capture human intent.

Trusted Display. A trusted display is necessary to deliver proper content to the physical user, given UI can be tampered by malware. Seeing properly rendered content is necessary for the user to correctly perceive the state of the computer and carry out actions following the user's original intent. Some work aim to protect the graphical stack. For instance, Trusted Display [119] uses a hypervisor to isolate the GPU driver, TrustGraph [80] and B3 [38] that builds trusted graphical subsystem for high assurance systems. However, due to the complexity of the graphical subsystem, this approach is not feasible for commodity systems.

Other works use TEEs such as Intel SGX [39] and AMD TrustZone [66, 67] to achieve a trusted display. These works are platform-dependent, and rely on certain security indicator for the user to differentiate trusted from untrusted displays. As shown in other works [36, 86], the effectiveness of security indicators remains questionable, thus it is possible to cover the display except for the trusted display window and still affect user perception similar to Context Hiding attack [44].

Different from all previous approaches, AINT does not need any trusted display, it uses a outsource-and-verify approach to ensure user sees UI correctly. The rendering work is outsourced to the malicious OS and AINT only verifies the correctness of the rendering. This allows AINT to leave complex rendering code out of its trusted computing base.

Trusted IO. This set of work ensures the integrity of hardware IO, which is necessary if hardware IO reflects human intent. Overhaul [81] and AUDACIOUS [89] assume that the userspace application is malicious and may impersonate the user to access sensitive sensors such as GPS sensor or microphones. Their solution is to implement an OS-enforced ACGs [91], so that user interaction with ACG cannot be intercepted by any userspace application. Aware [85] binds operation to the UI so that sensitive requests can only occur from predefined UI. If the OS is tampered, a communication channel between the hardware and some isolated execution environment is needed. For instance, DriverGuard [21] sets up a channel between the hypervisor and a protected application in userspace, communication data between the two parties are encrypted. However, their trust model is self-conflicting, as the kernel is not

trusted but a driver inside the kernel is. Trusted Path [122] and SGXIO[114] uses a trusted hypervisor to provide trustworthy IO data to a userspace party. Similarly, Aurora [69] uses SMM to do the same thing.

AINT uses a hypervisor to intercept authentic network inputs, and process it inside a trusted execution environment.

2.3.2 Capture and Deliver Human Intent to a Remote Server

This category focuses on delivering human intent to a remote party assuming an OS-level attacker on the local client. Each work has a different definition of human intent. Binder [29] and NAB [51] leverage *timing* as a heuristic to guess whether a network request is user intended. NAB enforces the weakest check, it only ensures a network request can only be sent shortly after a hardware IO by the user, e.g. keyboard press. Since the hardware IO is not tied to the outgoing request, an attacker can harvest user activity to generate tampered requests. Binder implements a stronger check for intrusion detection. AINT captures human intent from displayed inputs and binds the inputs to the request by directly generating the request from the inputs.

Gyrus [59] also leverages displayed inputs, and assumes that a user ensures the inputs are displayed correctly whiteout tampering. It uses a hypervisor to ensure that the content of outgoing packets matches the on-screen text. There are several problems with their approach. First of all, the service request cannot be assumed to contain the same on-screen text, due to client-side processing such as encryption. Secondly, Gyrus places too much burden on the user. For instance, Gyrus requires the user to validate everything user enters in a field, this may not be possible if the user is composing a long email. And lastly and most importantly, under a OS-level attacker, the rendering of applications cannot be assumed to be correct. There are many attacks that target users and attempt to affect human intent by modifying the UI, as shown in Section 2.1. To give an example, assuming Alice wishes to transfer $100 to Bob through email transfer, she does not know Bob's email address, thus she relies on some address book service to retrieve such information. Assuming the memory is protected, a rootkit can stealthily change Bob's email address shown on the screen to Mallory, and Alice would retrieve Mallory's email instead of Bob's, and carry out an transfer to Mallory instead. In this example, the problem is the UI of the address book service is tampered to affect Alice's perception; and Gryus cannot do anything to prevent this type of attack since it assumes Mallory's address is Alice's intention. In conclusion, under an OS-level attacker, on-screen text cannot be assumed to reflect human intent because the display may be tampered or even entirely crafted to affect user perception.

VButton [66] and Fedelius [39] deploy a trusted display embedded inside a larger untrusted display. Any user IO with the trusted display is guaranteed to come from (in the case of output) or go to (in the case of user input) a trusted execution environment (TEE). You can think of a TEE as a secure area of the processor with integrity and confidentiality protected. An example that Fedelius gave in their paper is that, on the payment page of a shopping website, the credit card form is rendered inside a trusted display, while the rest is rendered by the malicious OS; a user entering credit card information through the trusted display is secure. It is difficult to make the entire screen trustworthy because rendering engine that handles screen output and drivers that handle user input have gigantic code base, and large code base reduces the security guarantees. Also, large code base may not be feasible to run in some TEEs such as Intel SGX due to performance overhead and lack of system call support. However, this method is not secure either, an attacker can carefully craft the untrusted and unprotected part of the display and

still affect user perception [44]. For instance, Alice may think that she is interacting with Amazon.com and by entering her credit card information through the trusted window, she will pay for the goods in her cart. But in fact, a rootkit has invoked a different website attacker.com, and disguised the UI of attacker.com underneath amazon.com, the credit card form, where Alice believes to be Amazon's, actually belongs to attacker.com, and if Alice continues the transaction, will pay to attacker.com instead of Amazon. The above two examples show that physical users are vulnerable to UI attacks, given UI attacks can be easily carried out by OS-level malware, we believe current defenses are inadequate.

All works in this section are vulnerable to CF attacks; CF has shown that users can be tricked, and their behavior does not necessarily reflect their intention. AINT only uses the display of user IO as human intent *after* validating the display is tamper-free.

Multi-factor Intent Verification. Multi-factor Intention Verification is similar to the commonly known multi-factor authentication in that the intention verification process involves another trusted entity. The trusted entity can be a separated device, e.g. cellphone, USB key or a trusted hardware component residing on the same machine, e.g. Intel TXT. In the first case, any commercially available multi-factor authentication that not only authenticates the user, but also involves the confirmation of the request content can be considered as multi-factor intent verification. Research work such as ProtectiON [32] and IntegriKey [33] leverages a standalone USB device that collects and send raw user IO to the service using an out-of-band communication channel for intent verification. In the latter case, Bumpy [84] leverages an IEE provided by Flicker to encrypt user IO data and then sends it to the services and UTP [43] provides an one-way trusted path from the user to a service using Intel TXT and a TPM.

Multi-factor Intent Verification aims to deliver raw user IO, but under CF attack, the user is tricked and her actions will not follow her intent. Therefore, multi-factor intent verification will not be able to guarantee the user's intent. Our work, verifies what the user sees, ensuring that the user's actions follow her intent.

Chapter 3

Context Forgery Attack(CF)

3.1 Threat Model and Assumptions

In this section, we properly define the shared threat model of two attacks: Context Forgery (CF) and OS-level User Impersonation (OS-UImp). This threat model also applies to the proposed defense AINT.

3.1.1 Attacker

We assume a powerful attacker with full control of the operating system (OS) and all applications, but not the hypervisor. The attacker has the ability to read, write any unprotected memory and execute arbitrary code with OS-level privilege. The goal of the attacker is to send network requests to the remote service without user's awareness.

Since AINT requires the use of processor-supported trusted execution mechanisms such as Intel SGX, we assume physical devices are trustworthy and that includes the processor, chipset and peripherals.

We do not consider denial of service attack because a compromised OS can always stop the user from making any request.

CF's Assumption and Focus. CF is a UI attack that only attempts to affect user perception through modification of UI. It assumes that 1) application memory is protected and 2) user inputs are protected. Therefore, CF attempts to lure the user into providing unintended inputs and genereate unintended service requests.

3.1.2 Remote Services

Since the service is one of the beneficiaries of AINT, we assume the service is trustworthy meaning that any content, e.g. web page, from the service is free of misleading information.

The mutual authentication between the user and service is orthogonal to our work. We assume the service deploys multi-factor authentication [3], so that the attacker can not simply steal user credentials and carry out an impersonation attack from another device. We assume the user authenticates the service with the help of server certificate validation [45]. We acknowledge that this does not prevent phishing attack, but we exclude phishing attack for two reasons 1) it conflicts with our prior assumption on *good* services, and 2) anti-phishing techniques is also orthogonal to our work.

3.1.3 Physical User

Since the user is also one of the beneficiaries of AINT, we assume a cooperating and non-malicious human user. We require user *cautiousness* so that the user will carry out only intended actions under a properly generated context. A careless user can claim any request to be unintended despite. Cautiousness includes the following

- **Natural reaction**: The user reacts properly to the presented context and only carried out intended actions. In other words, the user will not try to generate non-intended request or enter non-intended inputs.
- **Short-term Focus**: The user must ensure the display of her input matches her inputs through peripherals before moving focus to the next field. For instance, as the user inputs number 100, she ensures the display is 100 before moving on to the next input fields. We note that this is a common practice by users, as users generally ensure their typing is correct.
- **General Alert**: The user should also alert if noticing abnormal behaviour from the computer, e.g. jumping cursors, cursor moves in the wrong direction as user expectation.

When the user notices an anomaly, she should stop using her device altogether.

Our requirement of user effort is much lower than prior work such as Gyrus [59], who requires the user to validate all previously entered inputs in every field on the page and VButton [66], who requires tedious double confirmation before the request is generated.

3.2 Attack Overview

We identify Context Forgery(CF) attack, a class of UI attacks. We emphasize that CF assumes application memory and user inputs are protected, and tries to lure the user into providing unintended inputs directly through misleading UI. The main victim of CF are 1) the user who sent out unintended requests to remote services, as well as 2) remote services, that respond to the request and not knowing that they are not intended. This is particularly harmful to services where the liability is on the service side, such as credit card services.

The OS-level attacker in CF broadens the attack vector and makes current defenses fail. CF can achieve UI modification in several ways: 1) it can tamper with the bitmap in the display frame buffer or 2) it can tamper with the memory of the display driver stack, such as X server and OpenCL or 3) it can tamper with user space applications that is responsible for rendering such as the browser. We do not restrict how CF tampers the display. Even though UI attacks do not require an OS-level compromise, we think that there lacks of research on this type of attack. We emphasize that, in this thesis, the term *OS compromise* implies not only kernel memory compromise, it also includes any unprotected application memory compromise.

We compare CF with other UI attacks, phishing and clickjacking, as well as OS-UImp in Table 3.1. In this table, we adopt the most general definition for each attack.

Relationship of CF and OS-UImp. Context Forgery and OS-level user impersonation(OS-UImp) are parallel attacks, but they do share similarities: 1) they share the same threat model and 2) they both aim to cause unintended request to be sent to a remote party. Their difference is that OS-UImp aims to impersonate the user in sending out service request causing unintended consequences, while CF aims to trick users into sending out unintended request by themselves. Because these two attacks share

Attacks	Definition	Threat Model	Root Cause	Mitigation
Social engineering	A social engineering attack is an attack that uses social means such as deception and manipulation in order to gain access to information technology [40]		Human stupidity [40]	User education
Phishing	A remote attacker fraudulently acquire sensitive information from a victim by impersonating a trustworthy third party [58]	Remote attacker with no control on the local system	UI Deception	Deception detection and user education
Clickjacking	An attacker trick the user into interacting with UI elements of another principal, triggering unintended actions [54]	On desktop-based browser: a remote attacker controlled malicious website On Android: a remote attacker's application	1) UI Deception and 2) delivery of events to unintended recipient	UI tampering detection or prevent event delivery to unintended recipient
OS-UImp	An attacker attempts to craft new or tamper with existing network requests without user awareness [66]	An OS-level attacker able to tamper with any unprotected kernel or application memory	software systems are exploitable	built better software
Context Forgery(CF)	An OS-level attacker attempting use luring and deceptive UI to trick the user into carry out unintended physical actions	A local attacker with OS-level privilege on systems with application request protection and trusted path	1) UI modification and 2) trick the user to carry out unintended actions 3) use the unintended actions to trigger service requests	Detection and memory protection without relying on an OS

Table 3.1: Comparison of attacks mentioned in this thesis

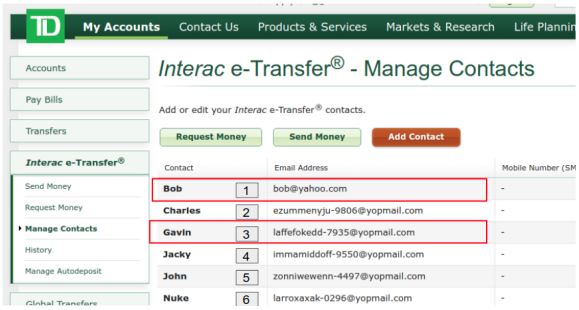


Figure 3.1: Address book service.

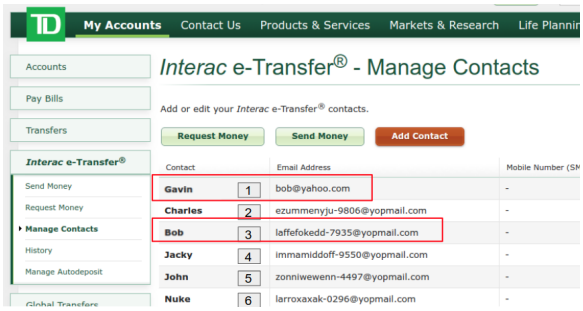


Figure 3.2: Tampered contextual information.

the same threat model, an OS-level attacker can easily carry out both. We point out that all existing solution that defeats OS-UImp suffer from CF, because of this, they all fail to provide the same security guarantee they claim (which is to send only user intended request to the service). In this work, our goal is to provide a solution that can defeat both CF and OS-UImp.

3.3 CF Examples

#1: Misleading Contextual Info. Imaging Alice intends to transfer $100 to Bob through email transfer on Paypal, while Mallory has OS-level privilege on Alice's client. When Alice visits the website, she relies on the address book service provided by Paypal to retrieve Bob's email address. Fig 3.1 shows the address book, where row 1 contains Bob and his email address and row 3 contains Gavin's name and Gavin's email address. Mallory can launch a CF attack by tampering with the name to email address mappings. To do this, Mallory can tamper with the display frame buffer or tamper with the rendering stack in the OS such as x server or GPU driver. The tampered UI is shown in Fig 3.2, row 3 contains the name Bob but is associated with Gavin's email address. Alice who intends to send money to Bob will click on row 3, which causes the computer to take the email address on row 3, Gavin's email address, and transfer money to Gavin.

In this example, the client-side rendering is being tampered. And thus, a defense is to validate the rendering of the address book against a proper rendering of the source HTML files.

#2: User Input Hiding. With the same scenario, where Alice wants to transfer money to Bob, Mallory can simply cause money loss by manipulating the display of transfer amount. As Alice types in 100(one followed by two zeros) in the amount field, Mallory intentionally blocks the display of the second zero, causing the digit '10' on the display, while the computer sees the field as '100'. Alice may be fooled by this number and thought that she mistypes, so she proceeds to enter another zero on her keyboard, making the display shows '100'. Her correction sends another zero to the computer, so the computer sees the field as '1000 instead of '100'. If Alice continues, a transfer of amount 1000 will be generated instead of user intended 100.

In this case, the display deviates from user inputs, and there is no validation of display against hardware inputs.

#3: Context Hiding. To ensure user perception is protected, prior works have adopted trusted displays embedded in an untrusted display for secure output of sensitive information. The trusted

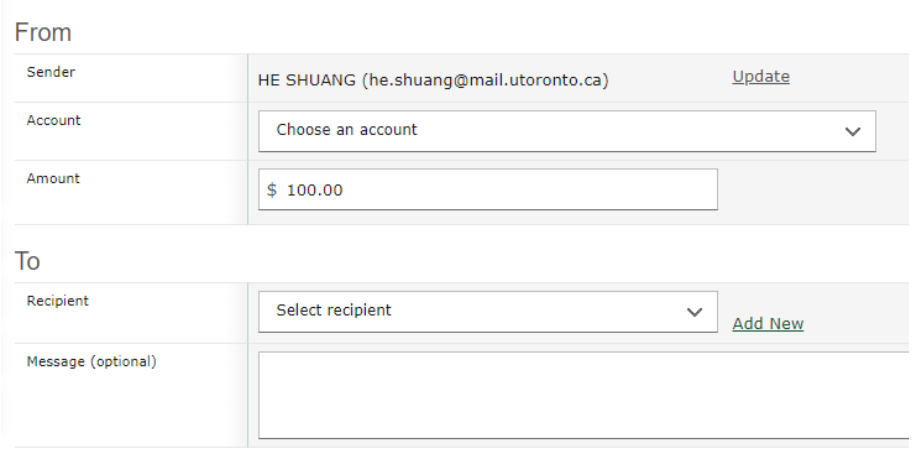


Figure 3.3: User sees 100, but a compromised OS can generate a request with a different amount.

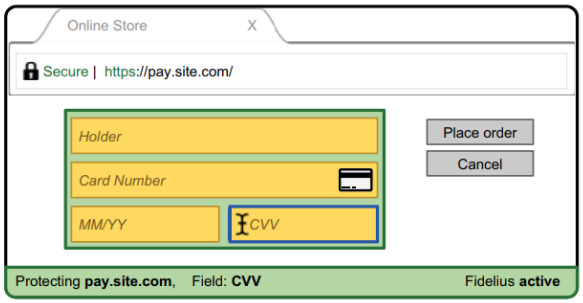


Figure 3.4: User interface of a trusted display. The green area is the trusted display overlay

display has a trusted channel to a secure environment on the computer allowing the user to enter inputs without the OS's inference. Translating this into our scenario with payment on Paypal, Alice will see the payment form rendered securely, and any inputs she enters will be securely captured. We borrow the figure from Fidelius [39] to show an example a trusted form in Fig 3.4. In this figure, Alice is only supposed to only interact with the green boxes. In the rest of this example, we aim to show that it is possible to affect user perception, even with the existence of the trusted display.

Even though Mallory cannot attack anything inside the green boxes, she can still launch a CF attack by crafting the untrusted display [44]. We show an example in Fig 3.5. In this example, the trusted display on the left side is what the web page generated and is the one that the user is supposed to interact with. Alice entering information in this form will safely pay for the goods in her cart. However, its semantic is altered to confuse the user.

The right side is also a real trusted form but is taken out-of-context from a different web session to the same payment service. Alice entering information in this form will pay $100 to Mallory. This form was taken out of context and put here with misleading and luring context — the attacker wants to lure Alice to interact with this window. If Alice interacts with the left side, she thinks that she is donating money, potentially resulting in payment amount not equal to the actual amount she needs to pay and cause unintended actions. If Alice interacts with the form on the right side, she pays to the attacker. The labels on the bottom of the page are supposed to help Alice identify which field she is interacting with, but it does not help in this case because both are legitimate trusted displays. The root cause in

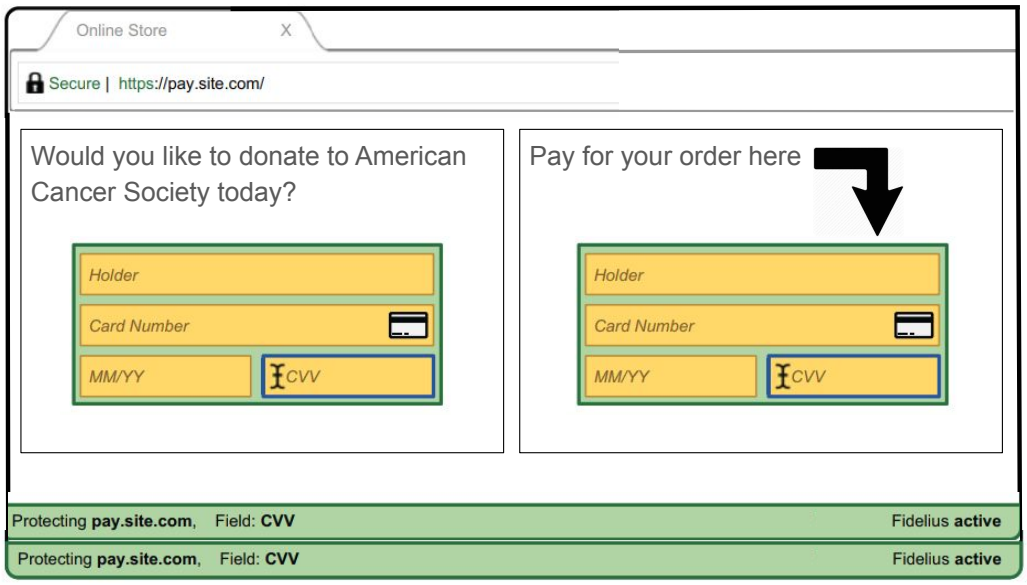


Figure 3.5: An example of CF that targets embedded trusted displays. The real trusted display is on the left side, but the attacker put confusing text next it to confuse the user its purpose. Another out-of-context trusted display is presented on the right side with misleading and luring contextual information.

this example is in two folds: 1) there is no enough information inside the trusted displays for the user to understand the consequences of their action. and 2) even though the trusted display is protected, the user still relies on the contextual information from the untrusted display. And since the untrusted display is not validated or protected, it can be tampered by the attacker.

3.4 Limitation of Current Defense

We discuss the limitations of current defenses of UI attacks and OS-UImp and contrast them with AINT.

Failure to resist OS-level UI attacks: current defenses for UI attacks rely on a properly functioning OS (on both x86 and Android). However, due to CF's OS-level attacker, none of them work.
AINT approach: AINT provides defense to OS-level UI attack without trusting the OS. AINT uses a trusted hypervisor for implementation.

Inaccuracy of intention inference: prior work [51, 29] infers user intention through heuristics such as time. The inferred intention is not accurate, allowing an attacker to bypass the checks.
AINT approach: AINT leverages user's Implicit Confirmation and uses user-validated on-screen text as user intention. Since any displayed text is user validated, and assuming a non-malicious user, AINT always captures user intention.

Extra user effort: Prior work [59, 39, 66] requires the user to perform certain extra tasks. However, a human user may not be able to perform these tasks.
AINT approach: AINT does not require the user to do extra work. It requires a minimum user effort where the user must validate the characters on both sides of the input cursor in the field where the user is interacting with.

Chapter 4

Design

Our objective is to develop a framework that can ensure the absence of UI tampering and prove to a service that a network request is user intended. Our solution must satisfy the following goals:

- **R1 Tamper-free User Interface (UI)**: the rendering of web pages is done correctly.
- **R2 Human Intent**: the content of the network request must be intended by the human user.
- **R3 No Indicators**: the effectiveness of security indicators is questionable [36, 86], we do not want any indicators.
- **R4 Minimum User Effort**: in minimum, the user should perform no extra effort.

The high-level idea of AINT is to "see" what the user sees: AINT records a video recording of the user's display. This recording shows what the user sees as well as how the user interacts with the computer. Therefore, to validate the rendering, AINT develops metrics such as image hash and optical character recognition(OCR) to ensure what the user sees is correctly rendered. To extract human intent, AINT relies on a property that we call Implicit Confirmation (IC), which states that human users validate the textual inputs on the screen against hardware inputs. Therefore, AINT extracts the textual inputs on-screen as human intent. Rich semantics can be inferred from the position of textual inputs and neighboring UI elements. To expose human intent to the service, AINT directly generates service requests using the captured intent. AINT requires the service to onlyare accept requests from client-side AINT because a malicious OS can craft its requests.

AINT is composed of the following components:

- **IntData**: This module captures authentic user IO data and user display (context) for future processing.
- **IntUI**: This module validates the UI is rendered properly according to the specifications from services.
- **IntInput**: Under the assumption of IC, this module tracks user focus and extracts user inputs and semantics from the display. The captured user inputs is treated as human intent.
- **IntRequest**: This modules binds human intent to the service requests by generating service requests with the human intent. It does so in a secure execution environment preventing execution tampering from a malicious OS.

We will discuss the design of these modules in the rest of the section.

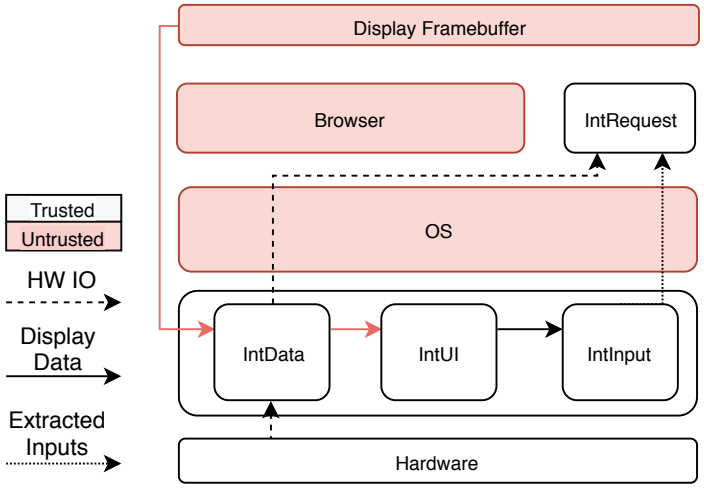


Figure 4.1: AINT Structure

4.1 IntData

AINT must capture user display and IO data for further analysis by other AINT modules. The captured data must be integrity and authenticity protected, ensuring that a malicious OS cannot tamper with the data captured and the captured data comes from the same machine as the user.

4.1.1 Context

Context consists of what the user sees and how the user interacts with the UI. To capture context, one may take screenshots, or sample the display, periodically. But an attacker can violate the *temporal integrity* by showing the user a tampered UI and quickly switch back to the proper UI before the sampling action. In this case, the proper UI was exposed for a small amount of time, and thus the change is unnoticeable by a human user. As a result, one may try to capture the context stealthily. But It is difficult to achieve stealthiness because there may be side channels where a malicious OS can use to infer periodic sampling. For instance, assuming an external HDMI grabber with USB connection to the machine, AINT can acquire user context by sending a command to the USB device. USB devices maintain a ring buffer for commands and a malicious OS can read from the ring buffer to infer the pattern where AINT samples the display, allowing the attacker to violate temporal integrity.

AINT samples the display with *randomness*. Specifically, AINT samples and waits for a variable time t (we refer to t as the interval variable) before the next sampling. This way, even if the attacker knows that AINT is sampling, she cannot predict when AINT samples. AINT still needs to decide on the frequency and randomness of the sampling process. On one hand, today's software frame rates have exceeded 60 frames per second(FPS), but sampling at a high frequency such as 60 FPS introduces too much storage and processing overhead (1 second of 60FPS recording with 1920 * 1080 resolution consumes 190 KB of space). On the other hand, AINT needs to sample the screen as frequently as possible to detect temporal integrity violations. Previous studies [112] shows the relationship between content exposure time and perceptual memory recall rate: with 500ms exposure time, a user can only recall 50% of the exposed content. This suggests that any tampered content can be exposed up to 500ms, while the user will only be able to recall 50% of the tampered content. Therefore, AINT uses 500ms

Browser	Code size (LOC)
Google Chrome[1]	25,670,051
Webkit[2]	17,145,901
Firefox[3]	20,548,088
AINT	2,436,717

Table 4.1: Compare AINT's user space code base with popular web browsers.

as the *exposure limit*. In other words, AINT assumes that any tampered content can be exposed to the user for a maximum of the exposure limit, and the user's perception will not be affected. To tune variable t, AINT samples variable t from a normal distribution with a mean of 250ms and a standard deviation of 83ms. This means that 99.73% of the time, the interval variable will be under 500ms, but with large randomness. We point out that this exposure limit does not prevent the user to alert to the change in the content — when the user notices a change in the content, she should alert and stop using the computer immediately.

4.1.2 IO Data

AINT requires the service to provide additional specifications that describe the expected appearance of the rendered web pages inside network packets. AINT must ensure that these specifications are not tampered by the malicious OS. To do so, AINT must capture network IO data with integrity guarantee. AINT leverages a trusted hypervisor to capture IO data using Intel Virtualization Technology (VT), which provides a method to trap to the hypervisor when the OS receives network packets.

4.2 IntUI

As pointed out by prior work on UI attacks [2], UI deception is the primary cause. Since AINT assumes trustworthy web pages from the service, the possible attacker vectors include all software running on the local OS. We define *service data* being the collection of data sent from the service for local processing, e.g. HTMLs, CSS and JavaScript.

To verify the rendering of service data, one straw man approach is to get it right in the first place: prevent a malicious OS from tampering with the rendering process. This solution is infeasible in several ways: 1) The rendering stack is huge and complex, it is difficult to get it all correct and bug-free. Prior work shows a linear relationship between code size and number of bugs [76], and the rendering stack consists of at least the GPU, GPU driver, OS and user space applications [77]. We compare the amount of code of popular browsers in Table 4.1 with the user space code in AINT. Despite the browser being only a single component of the rendering stack, its code base is huge. 2) Due to the size of the rendering stack, it is difficult to protect it entirely from compromises — it is infeasible to put the entire rendering stack in any trusted execution environment(TEE). As a result, there are solutions such as Fidelius [39] and VButton [66] that only put part of the rendering stack into TEEs implementing embedded truste displays. Even then, the TCB for these systems is not small: Fidelius includes two extra physical devices, each with a complete OS, and 8,000 lines of security-sensitive C code on the host; the TCB

[1] https://www.openhub.net/p/chrome/analyses/latest/languages_summary
[2] https://www.openhub.net/p/WebKit/analyses/latest/languages_summary
[3] https://www.openhub.net/p/firefox/analyses/latest/languages_summary

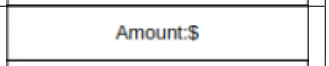
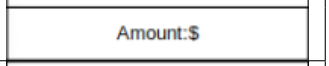
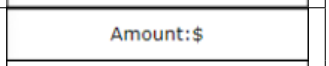
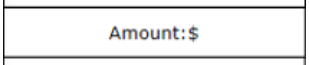
Platform	Browser	Cell Size W * H	Appearance
macOS 10.11.13	Chrome	199 * 34	
	Firefox	204 * 33	
	Safari	194 * 32	
Ubuntu 18.04	Chrome	192 * 34	
	Firefox	200 * 35	
Windows 10	Chrome	203 * 35	
	Edge	211 * 35	
	Firefox	211 * 35	
	Internet Explorer	197 * 35	

Table 4.2: Rendering Variations on different platforms

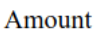
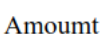
Appearance	Text	Cell Size W * H	Hash Value(Max=64)
	Amount	340 * 40	0
	Amoumt	340 * 40	7

Table 4.3: Image Hash fails to detect text tampering

for VButton includes a separate OS with a rendering stack. Moreover, these two solutions do not meet AINT requirement because embedded trusted displays need security indicators for users to distinguish trusted from untrusted displays. Also, embedded trusted displays can be exploited as shown by CF attack #3. In conclusion, a solution to validate the client-side rendering must be 1) robust to client-side rendering variations and 2) sensitive to subtle changes and 3) validates the entire screen.

AINT takes an outsource-and-verify approach, where AINT outsources the rendering to the potentially malicious OS and verifies the rendering is done properly according to specifications from the service. One method to do that is to calculate a cryptography hash value, such as MD5 or SHA, of the local display and compare it with *good* hash value from the services. Any difference implies modification. However, there are several problems: 1) there is a vast number of unique rendering stacks, and each will render the same content slight differently. For the same content, differences result in different hashes making it difficult for comparison. We show this effect in Table 4.2, the same content is rendered differently in different rendering environments. Due to the variation, it is impossible for a service to precalculate *good* hash values for all possible combinations of rendering stacks and 2) a cryptographic hash produces dramatically different results even with a single bit change in the input. Thus, it provides no robustness against local rendering optimizations such as font smoothing, e.g. ClearType on Windows [4]. Due to the pixel level difference, the same text will be hashed to dramatically different outputs even on the same platform with different software configurations, making it impossible to compare hashes.

Therefore, AINT uses an image hash. An image hash [110] is a hash function that hashes perceptually

[4]https://docs.microsoft.com/en-us/typography/cleartype/

similar images to similar hashes. By setting up a proper threshold, AINT can tolerate the rendering variations. But this also means that image hashes are not sensitive to subtle changes in the input image. For instance, a small change in the text can lead to a significant change in the semantics. We illustrate the inability in Table 4.3. The first row in Table 4.3 shows the original cell. The second row represents a tampered version of the cell. We evaluate these two cells using a Wavelet hash with 64bit output. We report the difference using Hamming distance. Our image hash thinks the two images are very similar, with 90% (7/64) similarities.

We need a solution that forgives the rendering variations but catches subtle changes. Our solution to this problem is in two ways: a design guideline for services and a validation method. The design guideline for website developers is web page tabularization, where the UI elements are organized into a tabular layout to enhance the machines checkability. The validation method bases on the guideline and works for individual cells. For every cell, to be sensitive and forgiving, AINT leverages both image hash and optical character recognition(OCR) that extracts characters off images.

4.2.1 Tabularization

AINT requires a web page to be tabularized into cells. The cells do not have to be aligned or have equal size, and are not directly tied to the underlining structure such as table cells in HTML. A cell in our abstraction is a unit of information, it contains either graphical, textual or input fields. The separation of cells makes up the boundaries of how much the hash function should check. We show an example of tabularized web pages in Fig 4.3. In this figure, we manually added the cell numbers in red cycles for illustration purposes.

We propose the following rules for tabularizing web pages:

- UI elements with text, such as labels, input fields, and interactive elements such as buttons, should be placed in its own cell, singled lined, with a clear background. This is to maximize the recognition rate of text, user inputs, and button labels. They take priority over graphical cells as graphical cells can be split.
- Avoid pure color cells by splitting graphical patterns. This is because image hashes are more sensitive to patterns and do not account for colors.
- Minimize the total number of cells for performance. We illustrate two different tabularization approaches in Fig 5.1 and Fig 5.2. We show the performance impact of tabularization in Section 6.

Web page developers have several ways to do tabularization. It can be achieved at HTML source level, or as a styling addition in CSS, or even as an image overlay to existing web pages. Therefore, we argue that the effort to convert a normal web page to a tabularized page is low. Specifically, the web page developer will 1) ensure every cell contains only one piece of information and 2) the page is fully-filled with cells. We claim that every web page can be converted to a tabularized web page.

Corner Encoding. The problem with tabularization is that the grid destroys user experience, making the pages unreadable. To reduce user disturbance, the grid is not directly shown to the user, the cells are encoded by their corners. We show the user's view of a tabularized web page in Fig 4.2. On the client-side, the grid is reconstructed before validation. The boundary between calls is implicitly inferred by the way services annotate pages: AINT uses a greedy approach to construct cells. If every four coordinates that can form a rectangle, then AINT treats it as a cell. Fig 4.3 illustrates the same web

page after cell reconstruction.

Benefits. Tabularization brings several benefits: 1) it divides the web page into many smaller areas so that the validation method can work on a smaller scope. In a smaller scope, subtle changes have larger impact — increasing the chance that image hashes will capture the difference. Tabularization also puts text into its cells, isolating other noises that can potentially confuse the OCR. 2) tabularization enforces a relative position on the cells that helps AINT locating information. For instance, in a money transfer form, the Amount label is placed on the left side of the Amount input box, allowing AINT to acquire the semantics of the numbers in the Amount input box. Without tabularization, the distance between the label and the input varies.

4.2.2 Validation Method

In this section, we detail AINT's method for validating client-side rendering. Our method applies two metrics: image hash and optical character recognition(OCR) to cells in a tabularized web page. Specifically, AINT checks 1) whether a hash computed from an image hash matches a good value and 2) whether text extracted from the cell matches the expected text.The good hash values and expected text are pre-computed by the service and shipped together with service data. When comparing calculated, AINT requires hash difference to be smaller than a pre-defined threshold while extracted text must be exact match. Rendering variations cause computed hash to deviate from the good hash, by allowing a threshold, image hashes can account for the variation.

AINT uses image hash to detect tampering similar to prior work [92, 23, 11, 101, 118]. but prior work requires the additional condition that two input images are already visually similar. In contrast, AINT has no prior knowledge on the appearance of the input images. As a result, image hashes in AINT have two problems: 1) similarity: two similar-looking images may be treated as one and 2) two distinct-looking images may be evaluated to the same hash [15]. In both cases, an attacker can replace UI elements with a different one and bypass the image hash check. We call images with same hash value but different appearance *alternatives* of the original images. AINT tackles this problem in two ways. Tabularization ensures that a single page is divided into multiple cells. And, it is difficult for an attacker to resemble several alternatives while still maintain a coherent semantics as a whole. Secondly for textual cells, AINT enforces both image hash and OCR checks, we argue it is difficult to change the appearance while maintaining the same text. We evaluate image hashes on these two properties in Section 6.4.

Unlike cryptographical hash functions such as MD5 and SHA, image hash provides none of the security guarantees such as first and second pre-image resistance and collision resistance, which makes image hashes vulnerable to the attack described above. However, image hashes are not replaceable by cryptographical hashes, because the latter provides no similarity check — similar but not identical images are evaluated to non-comparably different hashes.

The accuracy of OCR highly affects the accuracy of our validation method. However, OCR can achieve a high level of accuracy in our case for two reasons. 1) Tabularization enforces a textual cell to be separated from other content, reducing the noise in OCR. For instance, in Fig 4.3, cell #8, #10 and #11 are separated from the graphical content in cell #9, and thus allowing the content to be easily read. And 2) unlike hand-written letters, computer display fonts are easier to recognize. In our evaluation, OCR achieves high accuracy, but it does fail to recognize text occasionally. We discuss OCR in Section

Figure 4.2: Salt & Pepper: user's view of a tabularized web page

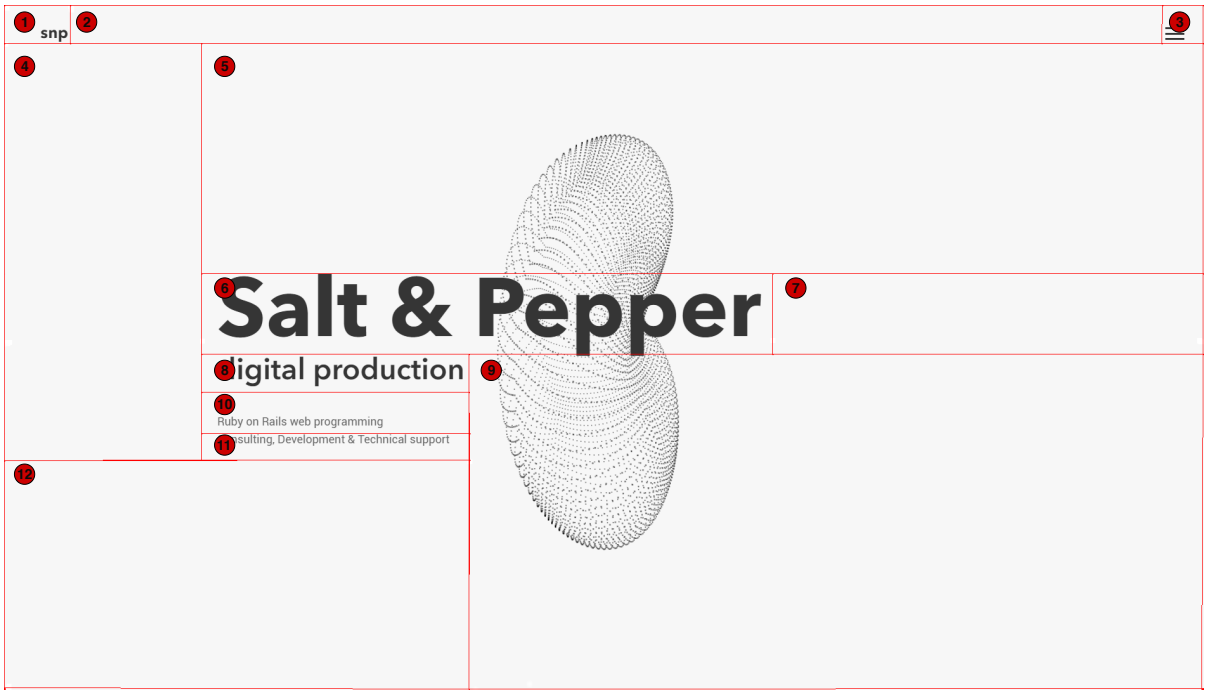


Figure 4.3: Salt & Pepper: AINT attempts to reconstruct the grid of a tabularized web page

6.2.

Validation Order. AINT only receives a list of *good* hash values for every cell on the page, but it does not know which hash corresponds to which cell from the captured context. Currently, AINT greedily reconstruct cells, and that allows AINT to assign a top to bottom, and left to right order to every formed cell, as shown in Fig 4.3. Therefore, AINT requires the service to order the hash values in the same way. Unfortunately, this also means that AINT requires a deterministic rendering on the client-side; the relative positions of cells cannot change. Either the user should disable browser features that may place the certain elements of the page in different places when resizing or the service should indicate to the browser that elements should not be moved.

Tamper-Free Tabularization. An attacker can tamper with the tabularization encoding and AINT can detect it. First of all, AINT knows the number of cells based on the number of good hashes. Therefore, any attempt to hide or increase additional cells will not work. Secondly, there is a chain effect if the attacker tampers with the size of a cell. Since the cells make up the entire page, if the size of a single cell change, the size and aspect ratio of the adjacent cells also change. This change propagates to all cells. Therefore, the entire tabularization scheme must change, increasing the possibility of AINT detecting the tampering. If the number of cells and the size of each cell remain the same, an attacker can only tamper with the cell's content, then the question becomes how does AINT ensures the validation accuracy. We argue it is difficult to tamper with textual cells, as the text must match. For the accuracy of the image hash, we refer the reader to Section 6.4.

Limitations. The hash calculation will be difficult if the entire display is not visible to the user and AINT. This includes pages out-of-view for scroll-able pages and multiple windows on the display, and only a portion of the secure form is on display. AINT tackles this problem by learning the position of cells currently visible and calculates the partial hash for the visible ones. Later, when the currently invisible cells are made visible, AINT calculates the total hash by combining previous partial hash with the hash values of the newly observed cells. To ensure a user fully understands the context on a AINT page, AINT will not generate service requests if the user has not seen the whole page. For dynamically loaded embedded content such as advertisement and iframes, the remote server does not necessarily know how the content beforehand, therefore, neither the design guideline nor the hash value can be enforced. We require advertisements and iframes to be excluded from the security-sensitive pages or the service has to enforce the same design guidelines on the content from third parties. Animations are static images that change over time. AINT is not designed to support the validation of animations, but one workaround is to check against a time series of hash values pre-calculated by the service and alert normally.

IntUI only validates the rendering of service data. System UI such as the cursor and the display of user inputs are not validated and are still vulnerable to CF. We address system UI-related attacks in the next section.

4.3 IntInput

The goal of this module is to acquire human intent. The machine only interacts with the human user through input/output(IO), and thus human intent must come from these two channels. Prior work such as NAB [51] says that since low-level hardware inputs, such as a keypress, can only be done by a physical human (assuming the general case with no hardware failure), they represent human intent. But

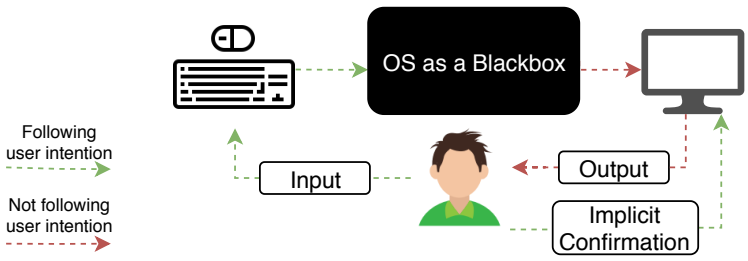


Figure 4.4: Implicit Confirmation

hardware inputs cannot be fully trusted as shown in CF attack #2. In that example, the user intends a transaction of $100, but observing the hardware inputs will show an amount of $1000. This difference occurs because the user observes that the number on the screen ($10) does not match her intention ($100). Therefore, she corrects the display by typing an additional zero. That example suggests that the display of user inputs on screen can be used as human intent because they are what the user sees and implicitly confirmed. Gyrus [59], is one such example, but it requires the user has to verify every input she typed from the entire screen at all times.

Therefore, the ideal method to acquire human intent is to combine the two by correlating displayed user inputs with hardware inputs. Specifically, for every character x of the inputs displayed on the screen, there must be some hardware input y that can justify the appearance of character x. This way, every input character on the screen is tied to certain actions by the human user. To give an example, when the display shows amount $100, there must be corresponding scan codes over the hardware wire. If there are more hardware inputs than what is shown on screen, then there is the possibility of CF attack. This approach suffers from the following aspects: 1) **Generality**: not all input methods leave reconstructable hardware traces. For instance, predictive text in recent Gmail web client allows a user to press tab to accept on-screen suggestions [25]. From hardware inputs, the observation is the press of the tab key, while the predicted text is not reflected. 2) **Trusted Code Base**: To handle this type of input method (i.e. predictive text, auto-complete and drag and drop) from hardware inputs, AINT must include the drivers and the software that produces these inputs. This is infeasible, because, in the case of predictive text, the code runs on the remote server, while in the case of drag and drop, it is the GUI Manager that handles it. Replicating the driver and the input method in the trusted AINT environment will blow up the trusted code base of AINT 3) **Synchronization**: After reconstruction, it is difficult to keep synchronized with the OS. For instance, an OS that is receiving updates of the cursor position from the mouse may skip updates from a certain interval due to system processing other events. A user sees cursor halting for a short interval, but this causes the states tracked by AINT to drift from the states tracked by the OS, as a result, any subsequent mouse position becomes invalid, and most importantly, AINT cannot distinguish such behavior is benign or malicious.

Since all three drawbacks are closely tied to the use of hardware inputs, AINT makes the design decision to not relying on hardware inputs to infer human intent. It solely relies on the context in which user interaction happens. Therefore, AINT assumes the user will react to any forged inputs and alerts.

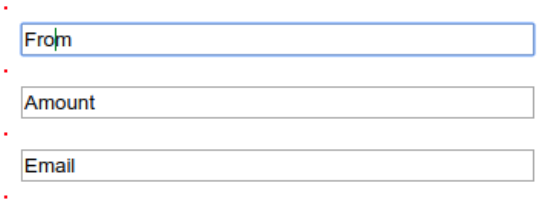


Figure 4.5: From field is currently under user focus, the others ones are not

4.3.1 Implicit Confirmation (IC)

AINT assumes the on-screen text represents human intent. A core insight is that non-malicious users will verify and correct on-screen text to be the same as their inputs through hardware. For instance, Alice who types 'abc' on the keyboard will verify that 'abc' shows up correctly — a practice commonly done already by users to check for typos. We call this process Implicit Confirmation (IC). We illustrate the concept in Figure 4.4. Assuming the user inputs what she intends, as shown by the green dashed line, as the data is passing through the driver stack inside a malicious OS, the OS can tamper with it and output arbitrarily on the display. This is indicated by the red dashed line. The displayed characters are implicitly confirmed. IC results in two possible states. First, when the display follows user inputs, no extra action is required. Second, when the display does not follow user inputs, the user will correct the display by correcting displayed inputs. This is illustrated in CF attack #2. In both cases, the result of the display is identical to human intent. We argue that IC holds as long as the user is careful. Since we envision AINT being used only with security-sensitive sites, the user is more likely to be educated about security and careful in their interactions.

The display is not protected from a malicious OS, thus the attacker is free to modify the displayed characters to trick AINT into extracting non-user-intended text. One straw man approach is to require the user to perform validation on the whole screen, all the time, but that is not feasible. To reduce user effort, AINT extracts user inputs as they appear on the display. To do so, AINT must figure out two things: 1) where the user is interacting and 2) how to extract the displayed characters.

4.3.2 Q1: Where is the user interacting?

To figure out where the user is interacting, AINT leverages the same method as to a physical human — focus box on the user interface. Modern web pages have a focus box that highlights the currently selected field. AINT searches for this visual indicator from the captured user context and takes that as the place where the user is currently interacting. We acknowledge that the style of the focus is page-specific, but we require AINT-enabled web pages to adopt a consistent design. An example of user focus is shown in Fig 4.5, the From field is currently under user focus, and the other fields are not under focus.

How does AINT and the user agree on a focus? Because an attacker can forge the focus box and cause the user's understanding of the focus box to be different from AINT's. If that is the case, the validity guarantee by IC will not hold. While it is attempting to broaden AINT's intake by adding code to eliminate any non-standard focus boxes, this method lacks generality as the definition of non-standard focus boxes is too broad and it is impossible to be exhaustive. Therefore, the approach AINT takes is to narrow the possible rendering of important system UI such as cursors, input cursors and focus boxes by *educating* the users about their appearance and perform *consistency checks*. Specifically, narrowing down

the system UI means that service should design cursors, input cursors and focus boxes with consistent appearance across platforms and browsers. And more importantly, services should acknowledge, train and educate users about the design. We note that this has been adopted in the industry. For instance, the search box on Google is consistent on different platforms.

With a limited number of possible appearances, AINT provides several consistency checks to reduce the common cases where system UIs may confuse the user. The consistent checks include the followings:

- **Number of cursors**: a cursor is defined as a movable indicator on the screen identifying the point that will be affected by the mouse input. Note that the cursor has different appearances: a regular pointer, a magnifier when zooming and a hand in drag & drop, all AINT supported cursors are shown in Table B.1. This check states that there is a maximum of one cursor, despite appearance, at any given time. Multiple cursors indicate a CF attack attempting to confuse the user.
- **Number of focus box**: a focus box is defined as the gradient around the textual input field when a field is under user focus. This check states that there is a maximum of one focus box at any given time on the whole page.
- **Number of input cursor**: an input cursor is defined as the blinking indicator inside a textual field that indicates the location of text entry. This check states that there is a maximum of one input cursor at any given time on the whole page.

It is important to note that the goal of these checks is to ensure that the user's perception of the state of the system is inline with AINT, rather than detecting any tampering on the system UI. Since these system UIs are maintained by the malicious system, it is impossible for an external observer, such as the user and AINT, to know whether their positions are correct with respect to all the historical hardware inputs. Consistency checks help AINT and the user to agree on the system UI. And, with this agreement, it does not matter how the OS interprets the input. To give an example, assuming Alice uses her mouse cursor to click on the Amount textbox and types in $100 in the field. The hardware inputs include the mouse and the keyboard. A malicious OS may interpret these values arbitrarily, by not moving to the Account field or show some number other than $100. If there were no consistency checks, the user may be looking at one mouse pointer while the AINT thinks the user is using a different one. With consistency checks, there is only one cursor and one focus box, and thus AINT agrees with the user on the semantics of these actions, e.g. clicking on the Amount textbox and filling out the value $100. The use will have to remain general alert to the following scenarios:

- The user should recognize the design style of the service and only interact with the legitimate UI elements.
- The user should alert when the machine reacts inconsistently to user inputs. E.g. inconsistent cursor movement or input display.

In conclusion, AINT enforces consistency checks on the UI so that AINT and the user can agree on where the user is currently interacting.

4.3.3 Q2: How to extract user Implicitly Confirmed inputs?

For any textual content, AINT can leverage OCR to extract user inputs from the display. But, how to reduce user effort from validating all inputs all the time to some level that is actually feasible? We observed that the user follows a linear pattern in entering and validating the text. Specifically, the user linearly goes over each field, and for each field, the user linearly enters inputs from left to right and linearly implicitly confirms the displayed characters. The linearity inspired us rules that can reduce user

validation effort: character order and field order. Note that our rules only describes a general pattern where inputs are entered, it does not prevent a user from going back to modify any inputs.

Field Order. Since AINT and the user agree on a focus box, changes in none-in-focus fields will be discarded by AINT. This means that the user only needs to focus on one field at a time and does not need to validate previously validated fields. If the user currently does not have a focus, and suddenly a focus appears, we assume the user will alert.

Character Order. Within a single field, it is difficult for the user to maintain focus on all prior entered text. For instance, as Alice is composing a long email, she can only focus on and ensure the correctness of the last five characters she typed. Therefore, AINT develops the following rules to reduce the user's effort in validating the text:

- **Left-side Insertion**: inputs can only be inserted on the left side of the input cursor.
- **Left-right Deletion**: inputs can be deleted from both sides of the input cursor.
- **Selection Focus**: if multiple characters are selected, they can be deleted at the same time.

The above rules restrict text changes to be near the input cursor while allowing input methods that insert or delete multiple characters at the same time. The exact number of characters that the user needs to validate depends on the input method: for manual typing that can only insert 1 character between two AINT samples, then the user only needs to focus on the validation of the character on the sides of the input cursor, because other characters cannot change. These rules significantly reduce the amount of work for the user to validate the text.

If any input moves out of the user's visible area, when the user is still typing, the user will not be able to IC. This broadly includes two cases, 1) when a cell scrolls outside of the view, and 2) within a field, the input overfills the currently visible area. In the first case, since neither AINT nor the user sees the cell, AINT will not update its copy of values. For the second case, if the input overflows the visible area, which could happen when the user copies and paste some long text. AINT cannot do anything in this case, because the text is never exposed to AINT. Therefore, AINT encourages services to design textboxes large enough to hold anticipated inputs and textbox overflow never happens. We argue that this is possible because AINT is designed to run on security-sensitive web pages such as a money transfer page, the service has a decent approximation in the length of the inputs.

And finally, AINT not only extracts ICed inputs, it also acquires the semantics of them by capturing their labels. Labels cells appear on the left side of the input cells. Thus, for every input, AINT forms a tuple $(label, data)$.

AINT waits for the application to indicate the end of an AINT session. Upon the signal, AINT will stop 1) performing consistency checks and 2) stop enforcing any rules. AINT checks whether the user has finished her interaction by whether the cursor falls in a specially marked tabularization cell, we call it *submit cell*. If that is the case, then AINT will invoke IntRequest to generate the service request. Otherwise, it considers the event as not being user-initiated, and more user inputs are there. It is possible for the OS to send an ending signal and fake the cursor location while the user is still interacting on the form. However, since there must be a change in the cursor position from its original position to the submit cell, we believe a careful user can notice such sudden change and can alert to such anomaly.

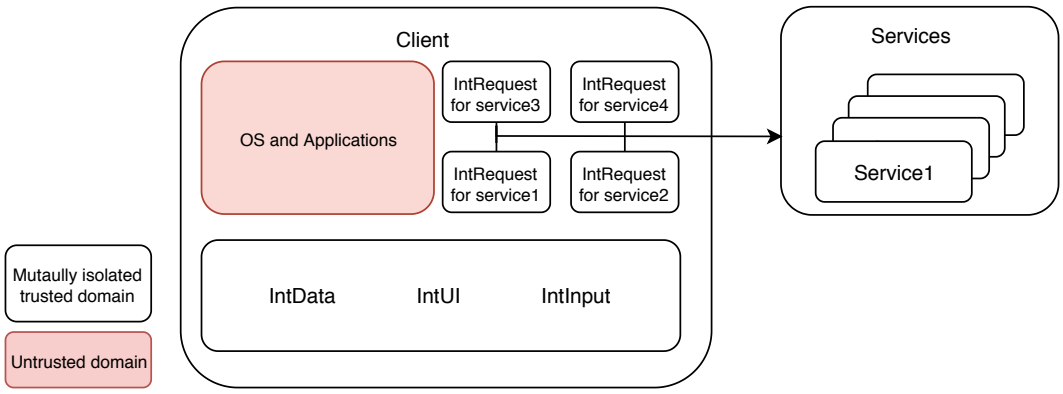


Figure 4.6: Trust and privacy boundaries

4.4 IntRequest

AINT provides an environment, called IntRequest, to execution service-specific code on the captured human intent. IntRequest is inline with the role of JavaScript in current web programming. Since an attack can tamper with the execution of any unprotected program, IntRequest must guarantee the integrity and authenticity of 1) IntRequest inputs and 2) IntRequest code and data. IntRequest leverages trusted execution environments(TEEs), which can protect the integrity and confidentiality of both code and data.

Because AINT does not restrict what the malicious OS can do, it might also generate service request to the service while AINT runs. Thus we require the service to only accept requests generated from AINT when it detects a client is AINT-enabled, and discard the request otherwise. For a service to know whether a client is AINT-enabled, AINT can do a remote attestation based on TPM [103]. On the service's side, this filtering can be implemented in a middle box using hardware or software [47].

AINT is designed with scalability in mind. We envision AINT serving multiple services, similar to the traditional client-server model shown in Fig 4.6. Each service designs its own IntRequest instance with customized logic. An IntRequest instance receives only human intent that the user designated to that service, e.g. the user entered 'abc' on Amazon.com will only be sent to the IntRequest belonging to Amazon. Because there can be multiple IntRequest instances from different vendors running at the same time, it is important to 1) isolate IntRequest from the rest of AINT, and 2) isolate instances from each other. The intuition behind this is that we do not want bugs inside one instance to affect the other instances or the entire AINT. Therefore, we design IntRequest instances to run a different trust domain. Recent advances on Trusted Execution Environment (TEE) such as Intel SGX services this purpose and can protect IntRequest from others.

4.5 Workflow

The overview of AINT is shown in Figure 4.7. In a typical workflow, an AINT session begins by the local computer receiving network packets containing AINT-enabled web pages ❶. The packets will be intercepted, parsed and protected by IntData, at the same time, an IntRequest instance will be set up ❷. AINT system will wait for the local OS to render the web page, once the rendering complete, the browser should send a signal to indicate the beginning of an AINT session ❸. We emphasize that this

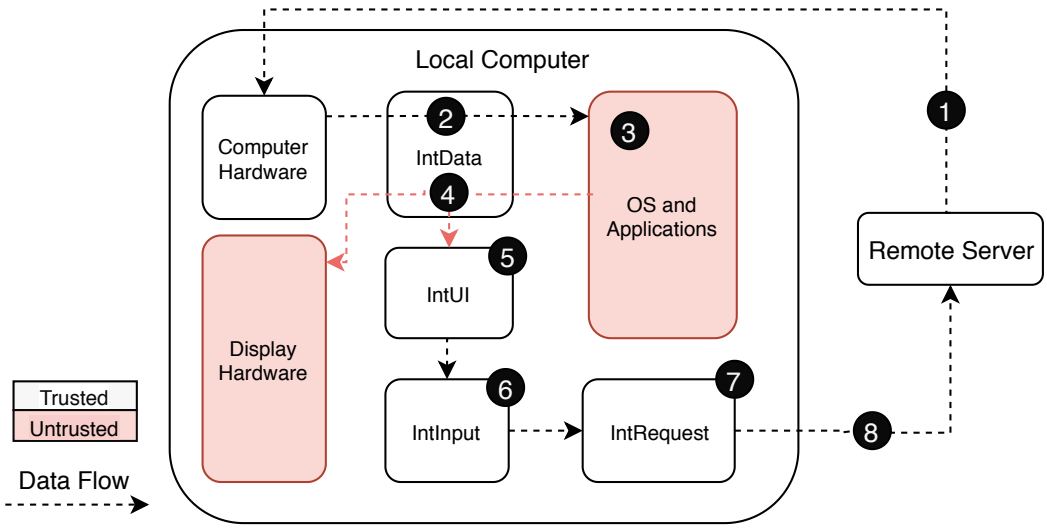


Figure 4.7: AINT Workflow

signal does not need to be trusted. Starting from the signal, IntData will capture user context ❹, IntUI will validate the rendering ❺ and IntInput will extract human intent ❻. When the user finishes her interaction with the page, the browser should send another signal. Upon receiving this signal, IntInput will send all collected human intent to IntRequest ❼ for the generation of a service request. IntRequest and the server will do a mutual attestation. IntRequest proof to the service the integrity of its code allowing the server to establish trust. IntRequest will execute service-specific logic, sign the result and send the result to the service ❽. The service only accepts requests signed by an AINT-enabled client.

Chapter 5

Implementation

We now describe in detail how AINT modules are implemented.

5.1 AINT-aware Service

Our design guidelines apply to services. We simulate an AINT-aware service by manually converting pages to use AINT. We converted several commercially available web pages plus one example page. The commercial web pages are Salt & Pepper as shown in Fig 4.2, two versions of Unhappy shown in Fig 5.1 and 5.2 and TD shown in Fig A.3 and Fig A.4, we also converted an example page called The Example shown in Fig A.1. We carefully selected these web pages because they do not have color variation, because AINT currently relies on a color-coding scheme for the detection input cursors, focus boxes and cello corners. This limitation exists to simplify object detection. We believe it is acceptable as a research prototype, commercial systems can implement object detection using other features such as the shape and position of UI elements. For every page, we go through the following procedures

- Tabularization: we tabularize based on our best judgment. Generally, we try to put each security-sensitive UI element in one cell. However, as shown in Fig 4.3, when elements is overlapping each other, one element must be divided. We use red dots to encode the corners of a cell.
- Unsupported features: we searched for animations, iframes and stepped view, and removed them since they are not supported.

5.2 AINT Hypervisor

IntData, IntUI, and IntInput are all implemented inside a trusted hypervisor. A hypervisor is a natural choice because it has a higher privilege over an ordinary OS: a hypervisor is generally considered as privilege level -1 while an OS kernel is considered at level 0. This privilege gives us two properties: 1) hardware-enforced *isolation* between the OS and the hypervisor, this protects hypervisors from a malicious OS and 2) hardware-enforced ability to *intercept* OS IO as required by IntData. Hardware vendors have developed accelerations for virtualization, such as extended page table(EPT) and Intel virtualization technology(VT). AINT's implementation takes advantage of these. Specifically, AINT requires EPT for fast guest linear address to host physical address translation, Intel VT for root mode (host) and non-root (guests) mode execution, as well as Intel VT-d for DMA protection over hypervisor

Figure 5.1: Unhappy: coarse-grained tablurazation

Figure 5.2: Unhappy: fine-grained tablurazation

memory in the case of direct memory access(DMA) attack by the guest. AINT can be implemented inside other low-level software environments such as Intel System Management Mode(SMM), Intel Management Engine(ME), as long as the environment satisfies the two properties. We choose hypervisor because it is the easiest to work with.

We experimented with several hypervisors to find the best fit. Since the hypervisor must be trusted, thus the trusted computing base(TCB) size matters. We tried XMHF [107], a formally verified research prototype micro-hypervisor with about only 6k lines of code (LoC), and Bitvisor 2.0 [96], another micro-hypervisor. However, there were several reasons why we did not use them. Specifically, for XMHF, we successfully intercept IO data (we were intercepting USB and PS2 keyboards at the time), however, to interpret the data, we had to move the entire driver stack into the hypervisor. For USB, we wrote a simplified USB 3.0 driver that reads the data packet. This approach is not generic because, for each type of IO data, we had to write a driver inside XMHF. Another drawback is that XMHF only supports 32-bit guest OS, our vision was to implement IntRequest inside SGX, which is only available on 64-bit, we attempted to port 32-bit XMHF to 64-bit, but it did not go well. Bitvisor supports 64-bit guest but we noticed that it crashes randomly. Both micro-hypervisors also have poor performance. XMHF is single-threaded, if there is a *vmexit*, all cores stop and wait for the working core to complete. In the case of Bitvisor, it does not use up-to-date hardware technologies such as Intel VT for fast guest logical to host physical translation — it relies on a shadow page table, which is a software implementation of the translation in the hypervisor. Also, both micro-hypervisors suffer from a vulnerability that hurts its isolation property on multi-core platforms [121]. We think that the effort to improve these micro-hypervisors out-weights their benefits. For this reason, we use Xen as the hypervisor, it is the best result in a trade-off between usability, performance and TCB size. It achieves good performance and is easy to work with than the micro-hypervisors. One characteristic of Xen that significantly benefits us is the use of dom0. dom0 is a specifically guest that is trusted and implements a split driver model. A split driver model is where all drivers for all guests are located inside dom0, and guest drivers are modified to use the driver stack inside dom0 for IO.

IntData. Recall that IntData captures the data needed for analysis, and that broadly includes two things: network inputs and context — a recording of user interaction. Due to the split driver model in Xen, the backend drivers inside dom0 provides service to guests(domU). Therefore, for network IO, we can simply use TCPDUMP to capture network packets for domU. We can filter out packets for dom0 because network packets from domU come from a dedicated virtual bridge. Any network package captured will be checked to see if it is AINT-enabled. Theoretically, for a page to be AINT-enabled, it must contain tabularized cells, for which the implementation can be at various levels, a list of hashes for each cell ordered in a pre-agreed order, for which one of them must be marked as the submit cell, and a $\langle code \rangle$ element that contains the request generation code. We manually converted several web pages to follow the tabularization style. The guest display is set up through a VNC connection to dom0, therefore, we capture the display through a VNC recording tool called VNCREC [1]. This setup is similar to Gyrus [59].

IntUI. IntUI is implemented inside dom0 using OpenCV 4.1.0, the goal is to validate the rendering based on the recordings against the ground truth information from services. IntUI reads the recording frame by frame and validates it against ground truth. Specifically, AINT attempts to detect cursors

[1] https://wiki.ubuntu.com/ScreenCasts/VNCREC

before doing anything else. AINT comes with default cursor templates for Mac and Linux; it will search for these templates pixel by pixel. The supported cursors are listed in Table B.1. Every frame is then preprocessed to reconstruct the grid from encoded cells. The gird is drawn by greedily matching rectangles based on the top left corner. An example of a pre-processed frame is shown in Fig 4.3. Then, AINT detects the grid edge and divides the frame into cells. The number and the order of the extracted cells must match the number and order of hashes encoded in the HTML. We use a top-to-bottom and left-to-right ordering, and we consider two points to be on the same line if their positions are within 4 pixels. AINT will attempt to 1) calculate an image hash using 64-bit Wavelet hash with haar Wavelet frequency mode and 2) acquire the text using PyTesseract, a Python wrapper for Google Tesseract [2]. AINT preprocess the image through a series of noise and border removal, color manipulation, rescaling and thresholding before image hash. We adopted Wavelet hash from ImageHash library [3], version 4.0. Before OCR, AINT attempts to detect the text areas [4], and only apply OCR is a text is detected. This method increases the success rate of OCRs. We used Tesseract version 4.0, and we configured Tesseract to 1) use both the legacy Tesseract engine and the LSTM engine 2) only look for upper and lower case English characters and digits and 3) use a dictionary for modern English. To improve the accuracy of OCR, AINT follows the guidelines listed on the Tesseract website.

To compare locally captured hash and text to ground truth information from services, AINT compares image hashes using Hamming distance [5] and compare text using exact string match. Hamming distance is acceptable [102] because each bit in an image hash is calculated individually. We define the threshold to be 30, because it gives us the best result. To interpret this number, since the maximum hamming distance for 64-bit value is 64, our choice of threshold is conservative as it allows close to half of the image to be different. We note that this number is specific to 64-bit Wavelet hash, and does not apply to other hash functions. In the future, we plan to try different hash functions and a smaller threshold value. For text comparison, we require the text to match exactly.

IntInput. IntInput is implemented inside dom0 using OpenCV 4.1.0. It detects user focus based on the blue focus box that occurs around the inputs fields, exactly as shown in Fig 4.5. This is the standard focus box used by Chrome. Therefore, to port the current implementation of AINT to other browsers, new support for the default focus box needs to be added. Currently, IntInput requires a green input cursor for the easy of detection. But this is not a hard requirement. AINT uses OCR to detect what the user has typed inside the focus box, similar to how AINT detects text using Tesseract.

Interface between OS and Hypervisor. AINT hypervisor must expose an interface for the browser to acknowledge the begin and end of an AINT session. The communication between AINT hypervisor and userspace application is implemented using *vmcall* instruction, aka the hypercall interface. This is done similarly to the application and hypervisor communication in TrustVisor [72]. A *vmcall* instruction is a special instruction that traps into the hypervisor, similar to the system call interface provided by the kernel. The hypervisor then validates the arguments and handles accordingly. We point out that this is the only communication interface that AINT adds to the existing interface.

[2] https://github.com/tesseract-ocr
[3] https://pypi.org/project/ImageHash/
[4] https://github.com/qzane/text-detection
[5] https://www.tutorialspoint.com/what-is-hamming-distance

5.3 TEE

IntRequest is currently simulated inside dom0 as a regular process in the userspace. The communication between IntRequest and IntInput is simulated using sockets, since dom0 is already trusted. Our original design was to run IntRequest in an SGX enclave and have the communication encrypted. We simulate a simple string concatenation inside IntRequest.

5.4 Cache

The performance of AINT can be significantly improved with caching. Since AINT does the repetitive work of validating the rendering of every cell in every frame, and most cells do not change over time: graphical and textual cells do not change at all, input cells only change while the user interacts with it, it remains constant before and after user interaction finishes. Therefore, AINT can cache the results of a prior validation and only validate if the appearance change.

Despite the high collision rate of image hash, both image hash and cryptographic hash are good candidates to calculate the cache keys. We used the image hash because it can potentially save the computation latter if validation is deemed necessary. On the other hand, a cryptographic hash 1) has a much lower collision rate and 2) it is hardware accelerated [56], thus providing better performance on key calculations. The choice between an image hash and cryptographic hash is specific to the user environment. In our implementation, we used the image hash as the cache key. We further explore the trade-off in Section 6.

Chapter 6

Evaluation

In evaluation, we aim to answer the following questions

- Q1: Whether AINT can prevent CF? That is whether AINT can detect tampering made to the web page.
- Q2: For actual usability, we ask whether AINT can tolerate rendering differences.
- Q3: Whether AINT can properly extract user inputs?
- Q4: How fast can AINT perform validation and intention extraction.
- Q5: Since image hash is critical to UI validation, we aim to find out how much difference can image hashes tolerate and their collision rate.
- Q6: What is the Trusted Computing Base (TCB) size of AINT?

The specs for the machines used in evaluation are listed in Table 6.1.

	CPU	Memory	OS	Kernel	Condition
CPU experiments	Intel i7 3770	16GB DDR3 1333MHz	Ubuntu 18.04	4.15.0-66	Hypervisor: Xen 4.9.2
GPU experiments	Intel i7 7700	16GB DDR4 2133MHz	Ubuntu 16.04	4.15.0-65	GPU: Nvidia 1060, 6GB GDDR5 OpenCL 1.2, CUDA 10.1.236

Table 6.1: Specification of machines used in evaluation.

6.1 Tampering Detection and Variation Tolerance

In this section, we answer the first two questions: Q1 Whether AINT can detect tampering on a web page and Q2 can AINT tolerate rendering variations?. The former is related to recall: a low recall rate suggests AINT's inability to validate the screen, which hurts AINT's security guarantees while the latter relates to precision, a low precision means AINT will not be able to tolerant variations and will cause usability issues. To evaluate recall, we simulated several UI attacks on The Example by manually modifying the HTML source, while for precision, we simulate the rendering variations on different platforms. We show the test cases and the result in Table 6.2 and Table 6.3. For all recall tests, AINT passes if it can print out an error message indicating the source of the error or the validation fails, while for precision tests, AINT passes if the validation passes. AINT cannot effectively deal with mouse cursors overlaying other objects, we address this failure in Section 7.

Test name	Description	Result
Tabularization	Removing tabularization dots or tamper with dot color	✔
Graphics	Additional graphical content added to a cell	✔
Labels	Single character change in a text label	✔
Input labels	Single character change in a input label, input label is the default text shown inside a input field, all text in Fig 4.5 are input labels.	✔
Structure	Additional box-like structure, simulating another input field	✔
Multiple cursors	Multiple cursors on the page	✔
Multiple focus boxes	Multiple focus box on the page	✔
Multiple input cursors	Multiple input cursors on the page	✔

Table 6.2: Tampering Detection

We believe that it is impossible to evaluate AINT's security guarantee by exhaustively trying all possible tampering and rendering variations. The test done in this section is for illustration purposes and justifies our prototype AINT works in the most basic case. The exact security guarantee depends on the generality of the detection mechanisms: OCR and image hash. We further discuss the security aspect of image hash in Section 6.4 and OCR in Section 6.2.

Test name	Description	Result
Input cursor detection and OCR	The blinking input cursor is intentionally placed beside a character with similar appearance (the character l)	✔
Cursor overlay 1	Mouse cursor over a button	✘
Cursor overlay 2	Mouse cursor over an input field	✘
Cursor overlay 3	Mouse cursor over a text label	✘
Font type	Serif font and sans serif font	✔
Font size	The text font size is rendered with its default size+2	✔
Color differences	the captured recording is compressed differently (using mov, mp4 and avi formats) causing color shift, e.g. pure red becomes darker red	✔

Table 6.3: Variation Tolerance

6.2 OCR Inaccuracy

This section answers the third question: Q3 whether AINT can properly extract user inputs? We were expecting 100% accuracy from the OCR engine, however, we notice that it produces unexpected results occasionally. We were unable to find a condition that can trigger OCR error consistently, except that the cursor overlay cases as shown in the previous section. To give an example, as the user types in '8759shuang@gmail.com', when it was still '87' on the screen, the OCR engine often mistakes the number 7 as a question mark '?'. Also, OCR occasionally mistakes 'gmail.com' as 'qmail.com' or 'email.com'. Since there is no concrete way to evaluate the accuracy of the OCR engine, we follow the best practice

listed under Tesseract website [1] and we detail the defensive mechanisms AINT deploys.

- **Cell pre-processing**. AINT reduces noise, enhances the character strokes, and turn images into grayscaled before invoking OCR.
- **Text location detection**. AINT attempts to detect the location of the text, and then calls OCR on the detected area.
- **Multiple calls** to OCR with different parameter. Tesseract with default configuration cannot detect single characters. AINT does repeated calls to Tesseract in case a single character is presented.
- **Multiple OCR engines**. AINT uses two engines from Tesseract, a legacy engine based on character patterns and an LSTM based neural net that focuses on line recognition.

Two problems arise from OCR Inaccuracy: 1) false positive in the tampering detection and 2) inaccurate extraction of user inputs. We detail some specific defense mechanisms.

- **False Positives in Tampering Detection**. AINT encourages fine-grained tabularization that clearly separates the text into individual cell, with clear background. However, we acknowledge that it is not always possible due to vertical text and world art as shown in Fig 5.2, or the background over text shown in Fig 4.2. It would be future work to better handle these cases.
- **Inaccurate Extraction of User Inputs**. AINT accumulates a history of user-entered inputs for every cell, and assuming OCR gives correct results most of the time, AINT can identify the mistakes OCR made and output a correct text in the end.

In conclusion, OCR in AINT implements mechanisms to achieve close to 100% accuracy on OCR. It ensures that AINT can properly validate web pages and extract user inputs.

6.3 AINT Performance

In this section, we aim to answer the question: Q4 how fast can AINT perform validation and intention extraction. Performance of AINT is important because it is on the critical path of the client and server communication.

Before we go into the numbers, we note that AINT is not designed to be an online tool. Its performance cannot keep up with the captured context. As the user interacts with the website, AINT only samples the context, the entire analysis and request generation are carried out after the session ends. This choice is made because 1) a lot of computation power will be consumed by the analysis step, AINT chooses not to downgrade user experience as the user interacts with the web page. To not downgrade user experience, AINT verifies the display at a later stage 2) due to the OS-level threat, even if the analysis detects anomaly, there is no particular countermeasure that AINT can do other than acknowledging the user and aborting the request. Therefore, analyzing at a later time achieves the same goal as if the analysis were done in real-time. We aim to see if the performance requirement of AINT is suitable for offline use.

6.3.1 Performance on Web Pages

The total performance overhead due to AINT is defined as $sampling\_delay + analysis + TEE\_overhead$. $sampling\_delay$ is the time between user ends an AINT session and when the sampling of that session

[1] https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality

is ready to be analyzed, this is essentially the execution time for IntData. *analysis* is the time between AINT starts validation and intention extraction and when AINT outputs the final user intention. It is the execution time of IntUI and IntInput. $TEE\_overhead$ is the slow down of running request generation code inside a TEE comparing to running it in a normal OS. This value depends on the specific TEE. In AINT, we use the trusted virtual machine, dom0, in Xen as the TEE, thus, we do not expect any $TEE\_overhead$. In this equation, we expect the total overhead to be dominated by *analysis*, and we evaluate the overhead of *analysis* in this section.

	Resolution (pixels)	Num of Cells	CPU Only (s)	w. Tesseract-OpenCL (s)
Salt & Pepper	1893 * 1080	12	7.57	6.388 (-15.61%)
Unhappy Less	1920 * 1080	4	3.503	2.598 (-25.86%)
Unhappy More	1920 * 1080	10	6.715	5.282 (-21.34%)
TD	1894 * 1080	58	30.861	25.809 (-16.37%)
The Example	1920 * 1080	9	4.17	3.36 (-19.42%)

Table 6.4: Seconds per frame for validating the AINT pages.

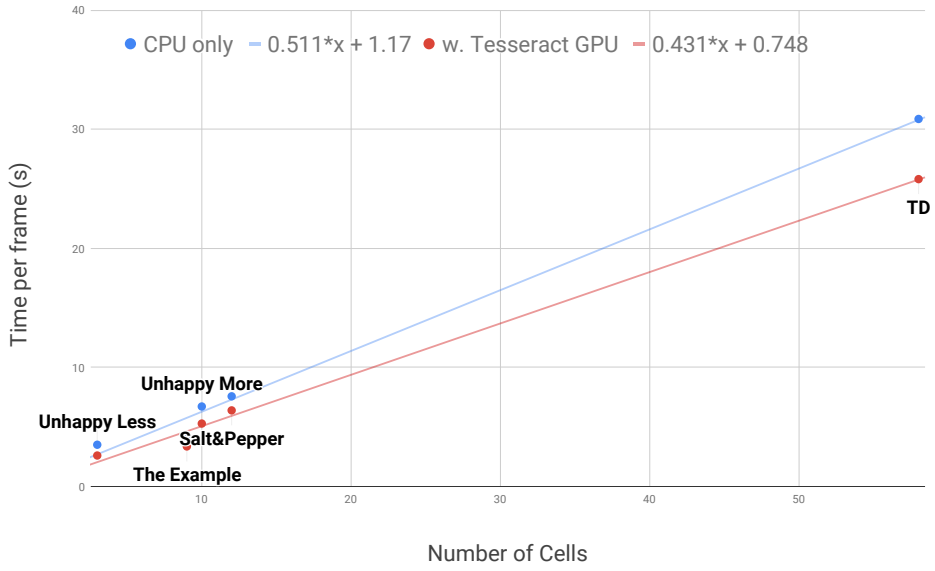


Figure 6.1: AINT Performance on web pages

We aim to figure out how fast AINT can validate the rendering of each frame. No user interaction is involved in this experiment. We report the information about tabularized web pages and the collected performance data in Table 6.4. We ran two AINT variations: CPU only and with Tesseract running in the GPU. From this figure, we see that for TD, it takes more than 30 seconds to validate a single frame, which means it is 120 times slower than without AINT. The performance of AINT does not meet our goal of 1 SPF and it is our future work to speed up its performance. We plot the performance in terms of the number of cells each web page has in Fig 6.1. From this figure, we see that there is a linear relationship between the total time and the number of cells. The reason is that for every cell, AINT needs to perform validation, and that includes computing the image hash as well as OCR.

Unfortunately, we were not able to run the two variations on the same machine and the GPU machine

has a faster CPU. We believe that it is acceptable because we do not aim to justify why the GPU set up is faster, we aim to figure out why the GPU set up has faster CPU and faster GPU and is not significantly faster than the CPU only set up. We see that, on average, GPU speeds up the performance by 20%. The reason why GPU is not as beneficial as expected is because AINT is written in Python while OpenCL and Tesseract naively support C++. This means that AINT must be written in C++ to take full advantage of GPU acceleration. In its current paste, every operation of Tesseract that attempts to use the GPU will have to go through 1) a transformation of Python data objects into OpenCL objects 2) a transmission from disk to GPU due to the use of PyTesseract and 3) transmission and transformation back. To improve the performance, AINT can be entirely implemented using C++, and take advantage of the GPU acceleration of both OpenCV CUDA and Tesseract OpenCL.

6.3.2 Micobenchmark

Function Name	Percentage of Overall Execution
Image Reading	1.5
Cursor Detection	25.22
Cursor Removal	0.68
Reconstruct grid	0.54
Cell extraction	1.77
Image hash	1.33
Text detection	0.37
Initialize Tesseract child process	6.54
Waiting for Tesseract to complete	54.52
Hash comparison & Sync extracted user inputs	0 (too small)

Table 6.5: Microbenchmark.

We aim to find out which part of AINT is the slowest: whether the bottleneck is related to the methodology or some external tool used by AINT. We profile AINT without caching for a single frame of The Example, and we list the percentage in Table 6.5. The microbenchmark was done using cProfile [2]. And we visualized the data by converting cProfile traces to kCacheGrind format [3] and pyprof2calltree [4]. We did our best effort to find components that spend the most time. But the total of our microbenchmark does not add up to 100% due to how kCacheGrind skips the functions that do not consume long.

PyTesseract and Tesseract. Tesseract is the OCR command-line tool, PyTesseract is a simple python wrapper to invoke Tesseract from Python. Tesseract is not performance-tuned, and PyTesseract adds additional overhead to it. Together, they make up over 60% of the execution time for a single frame.

Assuming a simple Tesseract function where it takes an image as input and outputs text in the string. By examining the code, PyTesseract inefficiently writes the image argument to disk and invokes Tesseract on that image through the command-line interface. Specifically, for each invocation, PyTesseract needs to write the image to disk, wait for this operation to complete, spawn a process to invoke Tesseract, set up the communication channel between the Tesseract process and the main process, wait for the operation to complete, read from disk the data returned by Tesseract and finally return to the caller. This process is inefficient, especially if done repeatedly.

[2] https://docs.python.org/3.2/library/profile.html
[3] http://kcachegrind.sourceforge.net/html/Home.html
[4] https://pypi.org/project/pyprof2calltree/

Tesseract itself is not performance-tuned. For instance, to be able to detect single character from an image, it requires a specially configured parameter $--psm10$. And to know whether an image contains only a single character, AINT checks whether the previous call to Tesseract returns an empty string. If that is the case, AINT calls Tesseract a second time with specially configured parameters in case if the image contains a single character. This causes repeated calls to Tesseract causing major performance overhead. There are also rumors on the Internet that suggests Tesseract is tuned for usability but not performance. Therefore, to improve OCR performance, a future version of AINT can use a performance-tuned OCR.

Template Matching. To exhaustively search for cursors, AINT checks the presence of the cursor at every possible pixel for every cursor type. AINT needs to check every cursor to know no duplicated cursors are presented. This process is inefficient, and as the result shows, makes up 25.22% of the overall execution. In our experiment on The Example, we had 7 types of cursors and 205200 (1920 * 1080) pixels in total. This repeated process of searching for cursors can be saved by caching.

6.3.3 Cache

	CPU Only	w. cache
Seconds per frame (seconds)	4.17	1.32
Number of cells validated	90	56

Table 6.6: Performance of AINT on 10 frames of 9 cells each and image hash as cache key.

Caching the results of validated cells should boost up AINT's performance, because AINT performs validation repeatedly over time. The exact amount of savings depends on 1) user activeness and 2) the design of the AINT web page. When the user does not perform any interaction, no cells change the appearance and thus AINT does not need to perform any validation. When the user interacts with the web page, only the cells that change need to be re-validated. A fine-grained tabularization scheme allows a larger portion of the web page content to be validated separately and benefit from the cache. We ran 10 frames of The Example on the CPU with the user filling out the form with cache. We list the average seconds per frame in Table 6.6. We see the average SPF drops from 4.17 to 1.32 using Wavelet hash as the cache key.

6.4 Image Hash and Cryptographic Hash

In this section, we aim to find out whether our hypothesis that image hash should return similar values for the *same content* rendered on different platforms and should return dramatically different values for *different content* is true.

There exists no comprehensive study on 1) similarity: how much similarity between two variations of the same image can be tolerated and 2) collision rate: what is the collision rate on images that have distinct looking. Translating these two properties to AINT. A high tolerance on similarity means that an attacker can assemble similar-looking and bypass the image hash check while a high collision rate means that an attacker can assemble differently looking images and pass the check. In both cases, a pure image hash-based checker will fail to detect any difference. In this section, we aim to answer the following questions:

- Q1: What is the degree of similarity that can be tolerated?
- Q2: Collision rate on distinct looking images.
- Q3: performance of image hash.

All evaluation is done on a machine with Intel i7-7700, 16GB of DDR4 2133MHz of memory. We use hash functions from ImageHash and cryptographic hash from HashLib on Python 3. For evaluation of Q2 and Q3, we evaluate these functions on Caltech 101 image dataset [5] from Caltech Vision Lab. This dataset contains 9144 images grouped into 102 categories, representing a diverse set of random images. We further define similar images in our dataset to be images from the same category, and random images if they belong to two different categories. The hash functions used in this test are Wavelet Hash with 8 bytes output (wHash 8), with 16 bytes output (wHash 16), perceptual hash with 8 bytes output (pHash 8) and 16 bytes output (pHash 16), as well as two cryptographic hash MD5 and SHA256, with 16 bytes and 32 bytes output respectively.

Image Hash Overview. We present how image hash works in general. Assuming an image hash with 8 byte(64 bits) output, the hash function will first convert arbitrarily sized (in terms of dimensions and the number of pixels) input images to an square-shaped 8 pixels by 8 pixels grayscale image, img_{input}. This conversion is necessary to match the output size — 64 bits in our example. Next, image hash will obtain an average representation of the image, img_{avg}. Usually, average representation is obtained by getting the median of luminance of img_{input}, sized to match img_{avg}. Depending on the image hash function, a transformation will be applied to img_{input} to retain the most representative part of the image. The transformation function includes Discrete Wavelet Transformation as used in Wavelet hash, Discrete Cosine Transform in Perceptual hash and a mean function in Average hash. We call the result $img_{feature}$. The final hash is calculated by flatten the result matrix of the difference between $img_{feature}$ and img_{avg}, e.g. an 8 by 8 comparison result is flattened to 64bit output. Thus, every bit in the output represents how a region of input image compares to the average perception of the whole image.

6.4.1 Similarity Tolerance

In this experiment, we designed three test cases of similar-looking images as shown in Table 6.7. The attack scenario is that, on a shopping page, an attacker may want to change the appearance of the product and trick the user to pay more than what they intend. For instance, paying the price of a modern beetle and get a vintage beetle instead, assuming the vintage beetle values much less. We evaluate image hashes on the two images and see if the hash difference is greater than the threshold. If the hash difference is smaller than the threshold, then it means AINT cannot detect any difference, which will disapprove our hypothesis.

We report the result in Table 6.8. For image hashes, we indicate the hamming distance between two hashes, while for cryptographic hashes, since the hash values are not comparable, we put a checkmark to indicate a hash difference. wHash 8 was used by AINT with a threshold of 30, and only beetles' hammering distance is greater than the threshold. This means that AINT will not be able to differentiate the two similar images in Greens and Cutlery. In other words, an attacker can interchange the two similar images and AINT will not be able to detect any difference.

To counter this, there are three ways: 1) a greater output size allows a hash value to capture more details of the input image, and thus, when comparing two similar-looking images, the difference will be

[5] www.vision.caltech.edu/Image_Datasets/Caltech101/

Beetles		
Greens		
Cutlery		

Table 6.7: Experiments used to test image hash on similar images.

larger. This trend can be justified by the increasing difference between 8 bytes output and 16 bytes output. 2) a different image hash function: on average, pHash reports a larger difference than wHash both on 8 bytes and 16 bytes output. 3) a better-chosen threshold. The threshold used in AINT was picked to allow the maximum variations of the same content rendered on different platforms (examples shown in Fig 4.2). This number may be too fit to the type of image in that experiment and not well suited for the types of images in this experiment. We leave the choice of a good threshold as future work. Therefore, for similarity check, AINT should have adopted pHash with 16 bytes output and a better-chosen threshold.

6.4.2 Collision Rate

A high collision rate on random images entails a high probability that an attacker can pick a random image and collide with the hash value of a given image. We conduct the experiment by running hash functions on every image in the Caltech 101 dataset and collect images evaluated to the same hash but from different categories (Inter-category images represent distinct looking images). We report the numbers of pairs of collisions in Table 6.9. This number is calculated based on $\binom{n}{2}$, where n is the number of unique categories within a list of images collide to the same hash. For wHash 8, there is a total of 77 pairs of images that hashes to 16 unique hashes. For wHash with 16 bytes output, pHash and cryptographical hashes, there is no hash collision.

	Beetles	Greens	Cutlery
wHash 8	39	25	7
wHash 16	104	96	36
pHash 8	28	27	12
pHash 16	160	118	117
MD5	✔	✔	✔
SHA256	✔	✔	✔

Table 6.8: Whether the hash values are different between the similar images. For image hash, we report the hamming distance.

wHash 8	wHash 16	pHash 8	pHash 16	MD5	SHA256
77 pairs of images in 16 unique hashes	0	0	0	0	0

Table 6.9: Number of pair-wise collisions of various hash function on randomly generated images.

We show some examples of random images with the same hash value in Table C.1 and we detail the limitations of wHash with 8 bytes output used in AINT. We observed the following: 1) different dimensions of the input image do not prevent collisions. As shown by the cougar face and motorbike example in Table C.1. Images with different dimensions look perceptually distinct to humans, but not to image hashes. We suspect that it is due to the resizing operation of hash functions. Specifically, all input images are resized to have an identical width and height. And because of this, long rectangular shaped input images loses more details compared to square-shaped images, because the former must be squeezed. 2) The shape of content contributes to the collision. As shown by the image with random backgrounds in Table C.1. Due to the circular shape, its hash collides with many other images with similar circular shapes. After resizing to 8 by 8, the details of the content are lost but the circular shape remains. 3) Image hashes do not account for colors. Even though different colors makes images perceptually different, colors are not part of an image hash. The random background image collides with the hedgehog despite their color differences. In conclusion, collision is relatively easy with wHash 8. AINT should have adopted other hash functions such as wHash 16, pHash 8 or pHash 16 to avoid collisions.

6.4.3 How to improve image hash functions for AINT

In this section, we propose how to modify image hashes algorithmically to better work with AINT. We leave the implementation of these as future work.

Currently, due to the constraint on the fixed output size, image hashes force resizing images with arbitrary dimensions to have a 1-to-1 ratio between width and height. We suggest an image hash that can dynamically allocate the ratio of the intermediate representations while still satisfying the output constraint. For instance, for a long rectangular shaped image with 160 by 40 pixels, the intermediate representation can be size 16 by 4, totaling 64 pixels. The output can still be 64 bits because both intermediate representations are 16 by 4.

Secondly, when converting images to greyscale, depending on the exact luminosity method used (our ImageHash library uses the ITU-R 601-2 luma transformation), it is rare for the resulting figure to utilize the full 0 to 255 spectrum. But if the spectrum can be fully utilized, meaning that the colors are distributed from 0 to 255, then more color information can be saved. To give an example, assuming the simple average luminosity method where the luminosity is calculated by $(R + B + G)/3$, if one image

contains pure red, green and blue only, the greyscaled image will have a constant luminosity (255/3). In this case, an image hash will fail to detect any feature and will produce non-meaningful outputs. An improvement can be made to use one end of the spectrum to represent red, e.g. 0, another end to represent green, e.g. 255, and have blue lies in the middle, e.g. 128. In this case, the full spectrum is utilized and the image hash will output meaningful values because it sees the difference in these colors.

6.4.4 Image Hash Performance

	Total (s)	Average (ms)	Total (s)	Average (ms)	Total (s)	Average (ms)
Output hash size	8 bytes (64 bits)		16 bytes (128 bits)		32 bytes (256 bits)	
wHash	33.14	3.62	33.47	3.66	36.10	3.95
pHash	8.617	0.94	10.55	1.15	16.58	1.81
MD5	n/a	n/a	2.52	0.275	n/a	n/a
SHA-256	n/a	n/a	n/a	n/a	3.91	0.43

Table 6.10: Performance of wHash and pHash for Caltech 101 varying hash size

6.5 Trusted Computing Base (TCB)

Modules		Lines of code (LOC)
IntUI and IntInput	High-level code	1,403
	OpenCV	2,053,651
	Tesseract	204,779
IntData	VNCREC	9,206
	TCPDUMP	167,678
Hypervisor	QEMU	1,625,674
	Dom0 kernel	17,193,756
	Xen	546,856

Table 6.11: TCB of AINT.

	TCB
Fidelius [39]	8k of rendering stack + two external hardware devices with full OSes
NAB [51]	A trusted hypervisor with hardware IO stack such as USB
aINT [6] [97]	A trusted hypervisor with hardware IO stack such as USB
VButton [66]	A trusted OS including a rendering stack
Gyrus [59]	A trusted hypervisor with network IO stack
Trusted rendering stack	A trusted OS with a rendering stack.
AINT	A trusted hypervisor, a minimum network IO stack and user space applications.

Table 6.12: TCB Comparison of AINT and other works

[6] A previous version of this work

This section aims to answer the last question: Q6, what is the trust computing base (TCB) of AINT? Prior work [76] has shown that large TCB leads to a larger number of bugs and thus weaker security guarantees. Even though the goal of AINT is to minimize TCB, its current implementation is not optimized for TCB. The TCB of AINT is detailed in Table 6.11, but the figures do not include Python libraries such as HashLib and ImageHash, which also uses other Python libraries, but we believe their code size is not comparable to OpenCV and Tesseract. The main components of AINT includes user space libraries and a trusted hypervisor. We list other works main components in Table 6.12. As shown in this table, a trusted hypervisor or OS is unavoidable. Note that, even with AINT current implementation, its TCB is smaller than to include a rendering stack into the trust domain (we consider a rendering stack being a trusted GPU driver and a browser, for which the former contains over 19 million LOC [7] and the latter contains over 25 million LOC [8]).

The majority of TCB comes from two parts: 1) userspace libraries, which includes OpenCV and Tesseract and 2) the hypervisor, which includes dom0 kernel and Xen. For the userspace libraries, we provide the following arguments and potential improvements. First of all, we use only a small portion of the libraries; not all code in the libraries are needed for AINT. OpenCV implements a tremendous amount of features, organized in 18 modules [9], but AINT only uses OpenCV for image manipulations such as contour detection, resizing and color changes. We believe that code minimization can significantly reduce the amount of code in those libraries. Secondly, these libraries are offline tools and do not communicate with the outside world. Therefore, they are hardly exploitable comparing to drivers. And lastly, our libraries run in userspace, for which should be isolated and protected properly by the hypervisor against a malicious guest OS. The large TCB for dom0 kernel and the hypervisor is due to our choice of hypervisor and the driver model of Xen. Rather than using Xen, AINT can be implemented with micro-hypervisors such as Bitvisor [96] and XMHF [107] to minimize the code base, which was our initial attempt, as discussed in Section 5. Note that, unlike previous work [97] that relies on a complex and error-prone USB stack [99, 22] inside the hypervisor, which is about 200k LOC, the only additional code required for a micro-hypervisor implementation of AINT is a network stack. A micro network stack such as uIP [34] contains less than 3k LOC. The TCB for IntRequest mainly depends on the type of TEE we choose. Our implementation of IntRequest does not further increase the TCB because we leverage the trusted VM(dom0) in Xen. If IntRequest is implemented using other TEEs such as SGX or TrustZone, then the TCB of the TEEs must be included.

6.6 Security Analysis

Hypervisor. It is attempting to argue that hypervisor is no more secure than an OS, and it is arbitrary to trust a hypervisor but not the OS. We argue that a hypervisor is safer than an OS: 1) hypervisor exposes a smaller interface. It exposes the hypercall and only traps on sensitive instructions, other times, the guest executes natively on the processor. 2) vendors like Intel provides hardware-assisted mechanisms to enhance the security of hypervisors. We envision that, in the future, the isolation between a hypervisor and its guests will be harder to breach. Addtionally, there is micro-hypervisors such as XMHF [107] and Bitvisor [96] who provide a high security guarantees. Therefore, we believe that it is possible for the OS

[7] https://github.com/freedesktop/drm-intel
[8] https://www.openhub.net/p/chrome/analyses/latest/languages_summary
[9] https://docs.opencv.org/2.4/modules/refman.html

to be compromised but not the hypervisor.

Initial Integrity of AINT Hypervisor. The hypervisor is assumed to be deployed during a trusted configuration phase before other software runs. This stage is commonly assumed in other security works. After boot, a remote party can establish trust in AINT hypervisor by challenging the local software. AINT can prove its identity and correctness by doing a remote attestation using the local TPM [103].

Runtime Isolation. AINT hypervisor must protect its memory from a malicious OS, this is done similar to how OS isolates userspace processes. Intel and AMD have developed Extended Page Table(EPT) that speeds up the translation between guest virtual address and host physical address. A hypervisor prevents a guest from accessing its memory by excluding its memory from EPT. For unauthorized direct memory access(DMA), Intel VT-d can be configured to protect hypervisor memory similar to EPT configuration.

Malicious OS temper with the AINT session indicators. A malicious OS may mess up the application's indication of the begin and end signals of an AINT session. AINT only generates a proper request if the user display matches the anticipated begin and end. The anticipated start of a AINT session lies anywhere between the page finishes rendering and the first user interaction. Sending a begin signal outside of this range results in AINT failing to validate the display, thus no request will be generated. The anticipated end of a session is when the user moves the cursor over a specially marked cell containing the submit button. Failing to meet this requirement results in AINT not generating the request. However, since AINT does not check whether a click actually happened, it is possible for an attacker to harvest the cursor movement over the cell and results in AINT generating requests with non-finished inputs. It is our future work to handle this case by correlating end signal with hardware inputs.

Chapter 7

Limitations and Future Work

AINT is only a research prototype, many things can be improved.

Image Hash and Threshold. As shown in Section 6.4, wHash 8 is neither sensitive to images with similar-looking but different content nor prevent collisions for distinct-looking images with different content. Also, the threshold was chosen based on one experiment. We plan to adopt pHash 16 with a better threshold that is picked through more experiments.

Automatic Tabularization. Currently, tabularization is done manually, it will greatly help the developers if this task can be done automatically.

Color-encoding. Currently, AINT relies on a color-coding scheme for some of its object detection. For instance, the cell encoding must be pure red, the focus box must be blue, and the input cursor must be green. Some of these assumptions such as the focus box come from real-life browsers such as Google Chrome, but not all assumptions will hold in real-life scenarios. Therefore, AINT's object detection can be improved by using more shape-based detection. For instance, focus boxes and input cursors only occur inside input fields, thus, they can be tied to the rectangular shape of the input field.

Support User Intention through other input methods. This thesis only deals with text fields while existing websites use other structures such as radio buttons and drop-downs. AINT can be extended to include these structures, however, the two tasks are to 1) validate the rendering and 2) extract the semantics. These two tasks have to be done individually for each structure and design carefully about how these structures interact with a tabularized web page.

AINT on Android. The entire work bases on x86, but it can be extended to Android. The entire AINT can be moved from a hypervisor into ARM TrustZone, a TEE that is commercially available on many ARM processors. Since the validation is designed for whole screen applications, AINT will fail if the user uses an on-screen keyboard, which is the primary input method on Android. One solution is to take the virtual keyboard into consideration and also validate the display of the keyboard similar to ScreenPass [70]. AINT focuses on web pages because AINT requires a way to gain ground truth information used for validation, and web pages files are sent to the client before rendering, thus providing a good opportunity to acquire this information. AINT can be ported to work with the traditional desktop application and Android applications, if there is a way to specify 1) the view that requires AINT to do checking 2) specifications used for validation and 3) what should AINT do after collecting user inputs.

This information must be sent to AINT with integrity and authenticity protection. Lastly, even though touch screen devices do not have cursors, but focus boxes and input cursors help Android AINT detect where the user focus is.

Cursor over text. Currently, AINT does not handle cursor over text well. When the mouse cursor overlaps with the text, AINT cannot detect the cursor, and OCR often mistakes the cursor as some character and gives inaccurate results. One solution is to leverage the GPU's help since we assume the hardware is trustworthy. Current cursors are hardware cursors where the GPU overlays the cursor image over the rest of the display. Therefore, AINT could obtain two video recordings, one with the cursor and one without. The validation would be done without the cursor while checking for request generation would be done on the version with the cursor. The performance and storage overhead of this approach needs to be determined. Running AINT on Android will not have the same problem because of the touchscreen.

Multi-page Support. Currently, AINT only works with a single page — the AINT session begins before the user interacts with the page and ends after the user finishes. AINT ensures user perception by enforcing Self-contained Context rule as specified in Section 4.2. This requirement can be dropped if the AINT adopted a similar approach as in reCAPTCHA, which uses user interaction throughout multiple pages to decide whether a final request is a user intended. For instance, to determine whether a user wants to submit the checkout for an umbrella on Amazon, AINT can check whether the user browsed raincoats, umbrella previously. This approach requires AINT to validate every site that user visits, extract semantic information and builds a model about potential user behavior. The benefit of this approach is the general applicability, as it works across websites, and reduces the burden on the user to do input verification, however, the model is probabilistic and may cause privacy issues.

Chapter 8

Conclusion

In conclusion, in this thesis, we showed that defenses for OS-level user impersonation attack (OS-UImp) fall prey to our identified user interface (UI) attack — Context Forgery (CF). We introduced CF and demonstrated several examples. The significance of CF is that it can be easily executed by the same attacker as OS-UImp and can cause the same damage: causing user unintended requests to be sent to the remote services.

We propose a defense AINT to both CF and OS-UImp. AINT has one module, IntUI, to validate whether the rendering of service data is correct according to specifications from the remote services. The main techniques used are tabularization and is to image hash and optical character recognition (OCR). AINT has another module, IntInput to acquire user intention. AINT assumes that a non-malicious user validates the display of user inputs, a process we called Implicit Confirmation, and the display input is the user's intention. AINT develops two techniques to help the user validate her inputs by 1) restricting the validation to the last characters entered and 2) restricting the validation to currently interacting fields. All captured are processed inside IntRequest instances with service-specific logic.

We show that AINT can detect subtle tampering and forgive rendering variations and its performance can be improved by caching previously validated tabularization cells. We went in-depth to evaluate the image hash we used, Wavelet Hash with 8 bytes output, on a more complex dataset: Caltech 101 image dataset. We showed that comparing to other image hash functions such as perceptual hash, Wavelet hash with 8 bytes output is neither sensitive to visually similar but different content images nor provide resistant to collisions on random-looking images.

We believe that AINT is a versatile tool for capturing and exposing human intent to services. It avoids a lot of the checks with sufficient but not excessive help from the physical human user. It minimizes the disturbance on the user and does not require the user to pay attention to any security indicators.

Bibliography

[1] Saeed Abu-Nimeh, Dario Nappa, Xinlei Wang, and Suku Nair. A comparison of machine learning techniques for phishing detection. In *Proceedings of the Anti-phishing Working Groups 2Nd Annual eCrime Researchers Summit*, eCrime '07, pages 60–69, New York, NY, USA, 2007. ACM. http://doi.acm.org/10.1145/1299015.1299021.

[2] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking revisited: A perceptual view of {UI} security. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[3] F. Aloul, S. Zahidi, and W. El-Hajj. Two factor authentication using mobile phones. In *2009 IEEE/ACS International Conference on Computer Systems and Applications*, pages 641–644, May 2009.

[4] AMD. AMD64 architecture programmer's manual volume 2: System programming, Sept 2018.

[5] ARM. Arm security technology - building a secure system using trustzone technology, April 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 689–703, Berkeley, CA, USA, 2016. USENIX Association. http://dl.acm.org/citation.cfm?id=3026877.3026930.

[7] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 97–113, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3341301.3359661.

[8] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 375–388, New York, NY, USA, 2011. ACM. http://doi.acm.org/10.1145/2046707.2046752.

[9] LLC BAE Systems Information Technology. Xts-400, Dec 2004. https://www.commoncriteriaportal.org/files/epfiles/st_vid3012-st.pdf.

[10] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, New York, NY, USA, 2010. ACM. http://doi.acm.org/10.1145/1755688.1755706.

[11] S. Battiato, G. M. Farinella, E. Messina, and G. Puglisi. Robust image alignment for tampering detection. *IEEE Transactions on Information Forensics and Security*, 7(4):1105–1117, Aug 2012.

[12] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

[13] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[14] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy*, pages 931–948, May 2015.

[15] Content Blockchain. Testing different image hash functions, Dec 2018. https://content-blockchain.org/research/testing-different-image-hash-functions/.

[16] Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure execution. *Technical report*, 2012.

[17] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Proceedings of the 2019 Network and Distributed System Security Symposium*, 2019.

[18] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, and Wenbo Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. 2007.

[19] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43(3):2–13, March 2008. http://doi.acm.org/10.1145/1353536.1346284.

[20] Yueqiang Cheng, Xuhua Ding, and Robert Deng. Appshield: Protecting applications against untrusted operating system. *Singaport Management University Technical Report, SMU-SIS-13*, 101, 2013.

[21] Yueqiang Cheng, Xuhua Ding, and Robert H Deng. Driverguard: A fine-grained protection on i/o flows. In *European Symposium on Research in Computer Security*, pages 227–244. Springer, 2011.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM. http://doi.acm.org/10.1145/502034.502042.

[23] Ondrej Chum, James Philbin, Andrew Zisserman, et al. Near duplicate image detection: min-hash and tf-idf weighting. In *Bmvc*, volume 810, pages 812–815, 2008.

[24] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM. http://doi.acm.org/10.1145/1966445.1966473.

[25] Bryan Clark. Gmail adds a predictive type feature called smart compose, May 2018. https://thenextweb.com/google/2018/05/09/gmail-adds-a-predictive-type-feature-called-smart-compose/.

[26] Graham Cluley. Viral clickjacking 'like' worm hits facebook users, 2010. https://nakedsecurity.sophos.com/2010/05/31/viral-clickjacking-like-worm-hits-facebook-users/.

[27] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.

[28] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM. http://doi.acm.org/10.1145/1814433.1814441.

[29] Weidong Cui, Randy H. Katz, and Wai-tian Tan. Binder: An extrusion-based break-in detector for personal computers. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Berkeley, CA, USA, April 2005. USENIX Association. https://www.microsoft.com/en-us/research/publication/binder-an-extrusion-based-break-in-detector-for-personal-computers/.

[30] Prashant Dewan, David Durham, Hormuzd Khosravi, Men Long, and Gayathri Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 828–835, San Diego, CA, USA, 2008. Society for Computer Simulation International. http://dl.acm.org/citation.cfm?id=1400549.1400685.

[31] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM. http://doi.acm.org/10.1145/1124772.1124861.

[32] Aritra Dhar, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. ProtectIOn: Root-of-trust for io in compromised platforms. *IACR Cryptology ePrint Archive*, 2019:869, 2019.

[33] Aritra Dhar, Der-Yeuan Yu, Kari Kostiainen, and Srdjan Capkun. Integrikey: Integrity protection of user input for remote configuration of safety-critical devices. Technical report, 2018.

[34] Adam Dunkels. The historical uIP sources, Dec 2018. https://github.com/adamdunkels/uip.

[35] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1065–1074, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1357054.1357219.

[36] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1065–1074, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1357054.1357219.

[37] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, July 2003. http://dx.doi.org/10.1109/MC.2003.1212691.

[38] J. Epstein, J. McHugh, R. Pascale, H. Orman, G. Benson, C. Martin, A. Marmor-Squires, B. Danner, and M. Branstad. A prototype b3 trusted x window system. In *Proceedings Seventh Annual Computer Security Applications Conference*, pages 44–55, Dec 1991.

[39] S. Eskandarian, J. Cogan, S. Birnbaum, P. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/SP.2019.00036.

[40] Nathaniel Joseph Evans. Information technology social engineering: an academic definition and study of social engineering - analyzing the human firewall. *in CONFidence*, 2011, 2009.

[41] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202. IEEE Computer Society, 2014.

[42] Facebook. Like button for iOS, September 2018. https://developers.facebook.com/docs/archive/docs/sharing/ios/like-button/.

[43] A. Filyanov, J. M. McCuney, A. Sadeghiz, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 1–12, June 2011.

[44] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057. IEEE, 2017. https://ieeexplore.ieee.org/document/7958624.

[45] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (ssl) protocol version 3.0. RFC 6101, RFC Editor, August 2011. http://www.rfc-editor.org/rfc/rfc6101.txt.

[46] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM. http://doi.acm.org/10.1145/945445.945464.

[47] J. Gil Herrera and J. F. Botero. Resource allocation in nfv: A comprehensive survey. *IEEE Transactions on Network and Service Management*, 13(3):518–532, Sep. 2016.

[48] Giorgio. Hello ClearClick, goodbye Clickjacking!, 2008. https://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/.

[49] GlobalPlatform. Tee system architecture v1.2, Dec 2018. https://globalplatform.org/specs-library/tee-system-architecture-v1-2/.

[50] Trusted Computing Group. Tcg d-rtm architecture, 2013. https://trustedcomputinggroup.org/wp-content/uploads/TCG_D-RTM_Architecture_v1-0_Published_06172013.pdf.

[51] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. Not-a-bot: Improving service availability in the face of botnet attacks. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 307–320, Berkeley, CA, USA, 2009. USENIX Association. http://dl.acm.org/citation.cfm?id=1558977.1558998.

[52] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM. http://doi.acm.org/10.1145/2451116.2451146.

[53] HTML5. Opera whole-page click hijacking via css, 2011. http://html5sec.org/#27.

[54] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 413–428, Bellevue, WA, 2012. USENIX. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/huang.

[55] Intel. Intel® trusted execution technology: White paper, 2010. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html.

[56] Intel. Intel® SHA extensions, July 2013. https://software.intel.com/en-us/articles/intel-sha-extensions.

[57] Intel. *Intel Server System R2600SR System Management Module (SMM) User Guide*, November 2017. https://www.intel.com/content/dam/support/us/en/documents/server-products/server-systems/R2600SR_System_Management_Module_User_Guide.pdf.

[58] Tom N Jagatic, Nathaniel A Johnson, Markus Jakobsson, and Filippo Menczer. Social phishing. *Communications of the ACM*, 50(10):94–100, 2007.

[59] Yeongjin Jang, Simon P Chung, Bryan D Payne, and Wenke Lee. Gyrus: A framework for user-intent monitoring of text-based networked applications. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, 2014. https://www.ndss-symposium.org/ndss2014/programme/gyrus-framework-user-intent-monitoring-text-based-networked-applications/.

[60] Karthick Jayaraman, Grzegorz Lewandowski, and Steve J Chapin. Memento: A framework for hardening web applications. *Center for Systems Assurance Technical Report CSATR-2008-11-01*, 2008.

[61] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *Seventeenth Annual Computer Security Applications Conference*, pages 265–276, Dec 2001.

[62] M. Khonji, Y. Iraqi, and A. Jones. Phishing detection: A literature survey. *IEEE Communications Surveys Tutorials*, 15(4):2091–2121, Fourth 2013.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1629575.1629596.

[64] Krzysztof Kotowicz. Filejacking: How to make a file server from your browser (with html5 of course), 2011. http://blog.kotowicz.net/2011/04/how-to-make-file-server-from-your.html.

[65] Eric Lawrence. Internet explorer 8 security part vii: Clickjacking defenses, 2009. https://blogs.msdn.microsoft.com/ie/2009/01/27/ie8-security-part-vii-clickjacking-defenses/.

[66] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 28–40, New York, NY, USA, 2018. ACM. http://doi.acm.org/10.1145/3210240.3210330.

[67] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[68] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 409–420, Philadelphia, PA, June 2014. USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin.

[69] Hongliang Liang, Mingyu Li, Qiong Zhang, Yue Yu, Lin Jiang, and Yixiu Chen. Aurora: Providing Trusted System Services for Enclaves On an Untrusted System. *ArXiv e-prints*, page arXiv:1802.03530, February 2018.

[70] Dongtao Liu, Eduardo Cuervo, Valentin Pistol, Ryan Scudellari, and Landon P Cox. Screenpass: Secure password entry on touchscreen devices. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 291–304. ACM, 2013.

[71] Luka Malisa, Kari Kostiainen, and Srdjan Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 289–300, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3029806.3029819.

[72] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society. http://dx.doi.org/10.1109/SP.2010.17.

[73] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1352592.1352625.

[74] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM. http://doi.acm.org/10.1145/2487726.2488368.

[75] Eric Medvet, Engin Kirda, and Christopher Kruegel. Visual-similarity-based phishing detection. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 22:1–22:6, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1460877.1460905.

[76] Subhas C. Misra and Virendra C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartI*, ICCSA'03, pages 724–732, Berlin, Heidelberg, 2003. Springer-Verlag. http://dl.acm.org/citation.cfm?id=1756748.1756832.

[77] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, pages 1–12, 2012.

[78] Marcus Niemietz. Ui redressing: Attacks and countermeasures revisited. *in CONFidence*, 2011, 2011.

[79] Marcus Niemietz and Jörg Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[80] H. Okhravi and D. M. Nicol. Trustgraph: Trusted graphics subsystem for high assurance systems. In *2009 Annual Computer Security Applications Conference*, pages 254–265, Dec 2009.

[81] K. Onarlioglu, W. Robertson, and E. Kirda. Overhaul: Input-driven access control for better privacy on traditional operating systems. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 443–454, June 2016.

[82] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 13–24. ACM, 2013.

[83] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[84] Jonathan M McCune Adrian Perrig and Michael K Reiter. Safe passage for passwords and other sensitive data. In *Proceeding of the 16th annual network and distributed system security Symposium*, 2009.

[85] Giuseppe Petracca, Ahmad-Atamli Reineh, Yuqiong Sun, Jens Grossklags, and Trent Jaeger. Aware: Preventing abuse of privacy-sensitive sensors via operation bindings. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 379–396, 2017.

[86] Rebecca S. Portnoff, Linda N. Lee, Serge Egelman, Pratyush Mishra, Derek Leung, and David Wagner. Somebody's watching me?: Assessing the effectiveness of webcam indicator lights. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1649–1658, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2702123.2702164.

[87] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1120–1136. ACM, 2018.

[88] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 945–959, Berkeley, CA, USA, 2015. USENIX Association. http://dl.acm.org/citation.cfm?id=2831143.2831203.

[89] Talia Ringer, Dan Grossman, and Franziska Roesner. Audacious: User-driven access control with unmodified operating systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 204–216, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2976749.2978344.

[90] Jeremiah Grossman (WhiteHat Security) Robert Hansen (SecTheory). Cursorjacking again, Sept 2008. http://www.sectheory.com/clickjacking.htm.

[91] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy*, pages 224–238, May 2012.

[92] Sujoy Roy and Qibin Sun. Robust hash for detecting and localizing image tampering. In *2007 IEEE International Conference on Image Processing*, volume 6, pages VI–117. IEEE, 2007.

[93] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Berkely, CA, USA, 1st edition, 2014.

[94] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2(6), 2010.

[95] Roger Schell, Tien F Tao, and Mark Heckman. Designing the gemsos security kernel for security and performance. In *Proceedings of the 8th National Computer Security Conference*, volume 30, pages 108–119, 1985.

[96] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1508293.1508311.

[97] He Shuang, Wei Huang, Pushkar Bettadpur, Lianying Zhao, Ivan Pustogarov, and David Lie. Using inputs and context to verify user intentions in internet services. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, pages 76–83, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3343737.3343739.

[98] Sean W Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[99] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM. http://doi.acm.org/10.1145/945445.945466.

[100] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association. http://dl.acm.org/citation.cfm?id=1298455.1298482.

[101] M. Tagliasacchi, G. Valenzise, and S. Tubaro. Hash-based identification of sparse image tampering. *IEEE Transactions on Image Processing*, 18(11):2491–2504, Nov 2009.

[102] Zhenjun Tang, Shuozhong Wang, Xinpeng Zhang, Weimin Wei, and Shengjun Su. Robust image hashing for tamper detection using non-negative matrix factorization. *Journal of ubiquitous convergence technology*, 2(1):pp–18, 2008.

[103] Trusted Computing Group. TPM 2.0 library specification, 2016. https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[104] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 645–658, Berkeley, CA, USA, 2017. USENIX Association. http://dl.acm.org/citation.cfm?id=3154690.3154752.

[105] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

[106] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.

[107] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 430–444, Washington, DC, USA, 2013. IEEE Computer Society. http://dx.doi.org/10.1109/SP.2013.36.

[108] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert Van Doorn. Carma: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 48–49. ACM, 2012.

[109] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 159–178, Berlin, Heidelberg, 2012. Springer-Verlag. http://dx.doi.org/10.1007/978-3-642-30921-2_10.

[110] R. Venkatesan, S. . Koon, M. H. Jakubowski, and P. Moulin. Robust image hashing. In *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*, volume 3, pages 664–666 vol.3, Sep. 2000.

[111] Tentacolo Viola. Cookiejacking, 2011. https://sites.google.com/site/tentacoloviola/cookiejacking.

[112] William von Hippel and Chris Hawkins. Stimulus exposure time and perceptual memory. *Perception & Psychophysics*, 56(5):525–535, Sep 1994.

[113] Helen J Wang, Chris Grier, Alexander Moshchuk, Samuel T King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *USENIX security symposium*, volume 28, 2009.

[114] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3029806.3029822.

[115] L. Wu, B. Brandt, X. Du, and Bo Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 55–63, Oct 2016.

[116] L. Wu, B. Brandt, X. Du, and Bo Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 55–63, Oct 2016.

[117] Min Wu, Robert C. Miller, and Greg Little. Web wallet: Preventing phishing attacks by revealing user intentions. In *Proceedings of the Second Symposium on Usable Privacy and Security*, SOUPS '06, pages 102–113, New York, NY, USA, 2006. ACM. http://doi.acm.org/10.1145/1143120.1143133.

[118] Cai-Ping Yan, Chi-Man Pun, and Xiao-Chen Yuan. Multi-scale image hashing using adaptive local feature extraction for robust tampering detection. *Signal Processing*, 121:1–16, 2016.

[119] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. Trusted display on untrusted commodity platforms. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 989–1003, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2810103.2813719.

[120] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. Sok: Hardware security support for trustworthy execution, 2019.

[121] S. Zhao and X. Ding. On the effectiveness of virtualization based memory isolation on multicore platforms. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 546–560, Washington, DC, USA, April 2017. IEEE Computer Society.

[122] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *2012 IEEE Symposium on Security and Privacy*, pages 616–630, Washington, DC, USA, May 2012. IEEE Computer Society. https://ieeexplore.ieee.org/document/6234440.

[123] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323, Washington, DC, USA, May 2014. IEEE Computer Society. https://ieeexplore.ieee.org/document/6956572.

Appendix A

Other Tabularized Web Pages

Figure A.1: The Example: an example of tabularized web page, user's view

Email transfer from your TD account	
From Account	From
Amount	Amount
Recipient Email	Email
Transfer!	

Figure A.2: The Example: AINT's interpretation

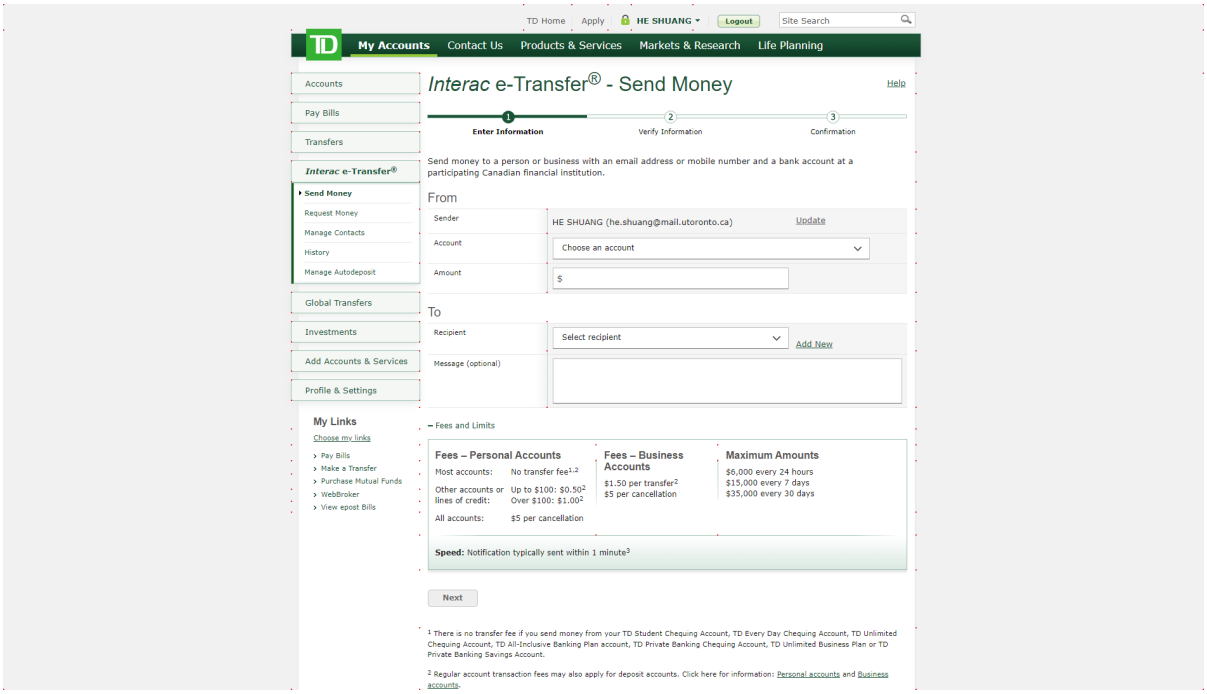


Figure A.3: TD: an example of tabularized web page, user's view

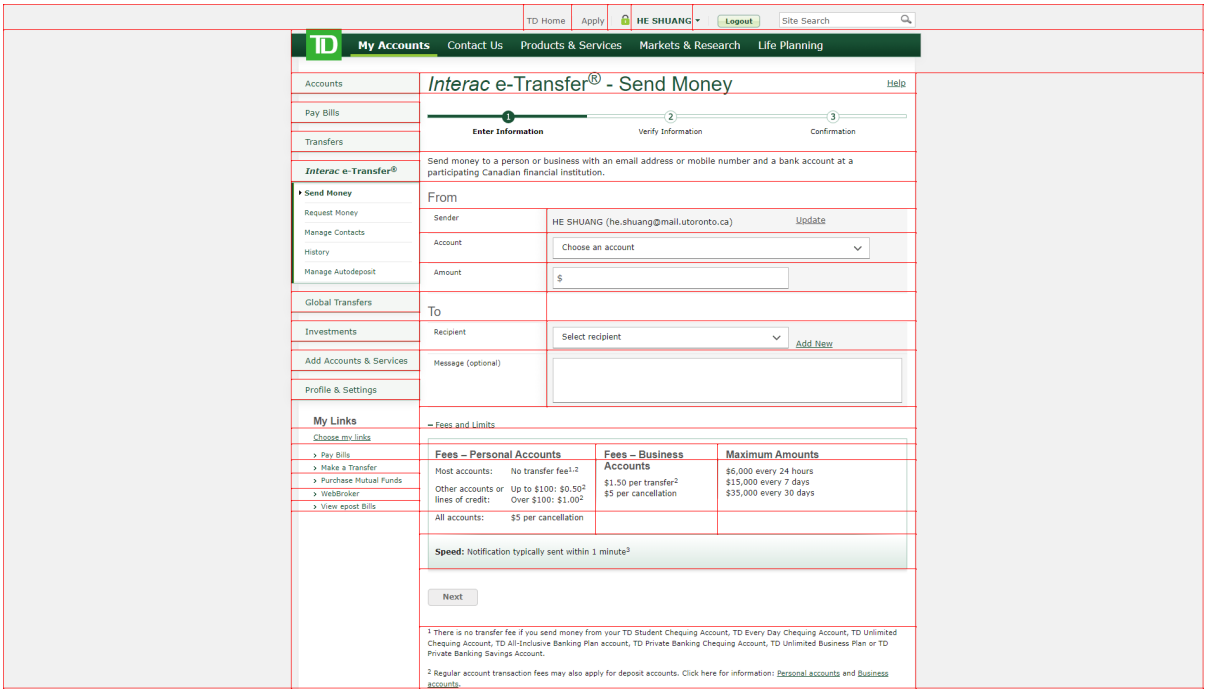


Figure A.4: TD: AINT's interpretation

Appendix B

Currently Supported Cursors

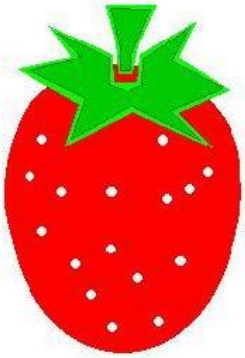
Cursor Types	Default	Drag	Option	Pointer	Type	Zoom-in	Zoom-out
macOS							
Ubuntu							

Table B.1: Currently Supported Cursors in AINT

Appendix C

Random Images Evaluated to the Same Hash in Caltech 101

Image	Image
ketch 	Joshua Tree 
Cougar Face 	Motorbikes 

Random Background 	Watch 
Random Background 	Umbrella 
Random Background 	Strawberry 
= Random Background 	Pizza 

Random Background 	Stop Sign 
Random Background 	Sunflower 
Random Background 	Hedgehog 

Table C.1: Each row represents a pair of random images from different categories.