

Using Context to Verify User Intentions

He (Shawn) Shuang

December 13, 2019

1 Introduction and Design

A malware on the user client¹ may generate service requests to a remote service without user's awareness. One possible defense is CAPTCHA. However, CAPTCHA falls prey to a more powerful attacker with operating system(OS)-level privilege. An OS-level attacker may tamper with program memory stealthily so that the service request generated from the memory is not what user intended. We call this type of attack OS-level user impersonation attack (OS-UImp). The state-of-the-art defenses [5, 6, 10, 13, 17] for OS-UImp try to revealed user intentions to the service. But they assume the user has a clear intention, perceives the UI correctly and performs the extra work as imposed by the individual solution correctly. We challenge this assumption. Prior work on UI attacks [12, 21, 23, 24] has shown the feasibility to affect user perception through UI modification. We identify a new class of UI attack called Context Forgery (CF), where an OS-level attacker modifies the UI to trick the user into carry out unintended actions which result in unintended service requests — the same consequence as OS-UImp. All of existing defense for OS-UImp fall prey to CF.

In summary, the limitations of the state-of-the-art defenses of OS-UImp are: 1 **extra user effort** is required and 2) vulnerable to **Context Forgery** attacks. 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, thus no tampering is possible. And to prevent user impersonation attack, by "seeing" how the user interacts with the page, we can infer her intention. By "seeing what the user sees", no extra effort is required on the user side. We illustrate our solution in the rest of this section.

Project Statement. We believe it is possible to build a solution to OS-UImp while 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, thus no tampering is possible. And to prevent user impersonation attack, by "seeing" how the user interacts with the page, we can infer her intention. By "seeing what the user sees", no extra effort is required on the user side. We illustrate our solution in the rest of this section.

We propose Attested Intention(AINT), a framework that defeats both OS-UImp and CF. AINT extracts the client-side display and validates it against specifications from services. Since AINT cannot rely on the OS, it proposes a design guideline, called Tabularization, for remote services to design better machine-checkable web pages. Tabularization requires the service to place UI elements into a tabular structure so that each cell can be individually validated. This method divides the difficult problem of validating the entire page into many simpler problems of validating cells. When a tabularized is rendered on the client side, AINT validates it using a combination of image hash and optical character recognition (OCR). Image hashes

hash visually similar images to similar values, and thus providing robustness to variations due to the rendering environments. OCR extracts text from images and thus provides detection on subtle tampering such as single character change. AINT requires the image hash difference between a local rendering and the good hash value to be smaller than a threshold, 30, and requires the text from the local rendering to match the expected text exactly.

With a properly rendered display, AINT infers user intention based on her interaction with the web page directly from the screen. 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. 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 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) and protects captured user inputs with isolation methods. When the user finishes her interaction, AINT checks user initiation based on the location of the cursor, cursor must be on the submit button for a service request to be generated. AINT supports the generation of the service request based on service-specific logic in a trusted execution environment (TEE). The generated request will be transmitted the designated service. AINT requires the service to be modified to only accept requests from AINT, a malicious OS can generate its own request, but it will not be accepted by the remote service.

We simulate our design guideline on five web pages. On the client-side, we implemented AINT Tesseract, OpenCV and Wavelet hash on top of Xen. We also added a cache in our implementation to speed up AINT's performance. We evaluate the sensitiveness and robustness of our validating method using various UI attacks and rendering variations we simulated. We show where AINT fails. 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 image hashes for tampering detection, 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.

2 Related Work

2.1 Clickjacking

One class of UI attacks is clickjacking attack, the idea is to overlay the victim application with attacker's UI, and while the user is thinking that she is interacting with the UI at the front, her actions such as clicks and taps are hijacked to trigger events in the victim application underneath [24] and cause unintended actions. Hijacking attack is not limited to clicks [22], it can be combined with other attacks such as CSRF [14], XSS [21] and CSS [11], to

¹In this project, we use the term *client* to denote any computing device, including smartphones and PCs.

steal files [15] and cookies [26], and with a bit of domain-specific knowledge, it can hijack likes and shares on Facebook [4].

The root cause of this type of attack is UI deception [1] and illegal delivery of user inputs [8]. To prevent UI overlay, Frame buster [16, 25] 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 [27] prevents cross-origin frames to be rendered transparently. This raises usability issues. To stop the delivery of user input to unintended 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 [7]. Another set of defense mechanism [2, 9], 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.

The main difference between regular UI attack and CF is the OS-level attacker. The main victim of CF is 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. CF can achieve UI modification in several ways: 1) it can tamper with the bitmap in the lowest level 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 the application level logic that is responsible for rendering such as the browser. We emphasize that CF is an UI attack, CF does not tamper with the application logic that generates network requests. CF assumes that part of the memory is protected [19, 20].

2.2 Deliver User Interaction to the Server

This category of work focuses on delivering user intention to a remote party assuming an OS-level attacker. Gyrus [13] leverages a trusted path for user IO and requires the user to validate the text values in the trusted system is what they intended. Then it uses a hypervisor to ensure that the content of outgoing network packets matches what the trusted system had. 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. An CF attack can trick the user into doing something she does not intends to do. And Gyrus does not provide any defense.

VButton [17] and Fedelius [6] deploy trusted displays embedded inside a larger untrusted display. The content and any user interaction in the trusted display is secure. However, it is difficult to make the entire screen secure because rendering engine that handles screen output and drivers that handle user input have gigantic code base. The embedded trusted display is not secure, an attacker can carefully craft the untrusted and unprotected part of the display and still affect user perception [8]. 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. But the credit card form, where Alice believes to be Amazon's, actually belongs to attacker.com, and if Alice

continues the transaction, will pay the money to attacker.com.

3 Evaluation

3.1 Tampering Detection

In this section, we answer the first question: whether AINT can detect tampering on a web page. We simulated several UI attacks on the AINT-enabled website The Example by manually modifying the HTML source. We show the test cases and the result in Table 1. For all the tests in this section, AINT passes if it can print out an error message indicating the source of the error or the validation of the target cell fails. AINT passes all tests.

3.2 Variation Tolerance

In this section, we aim to answer the second question: can AINT tolerate variations of different platforms. We simulate the rendering variations on different platforms. All tests and results are shown in Table 2. AINT passes all tests except the cursor overlay tests. We address this failure in Section 4. Recall that, in this section, a test passes if the image hash difference is less than the threshold, 30, *and* the extracted text is identical to the expected text.

3.3 AINT Performance

In this section, we aim to answer the question: 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.

3.3.1 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 3. 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.

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 the full advantage of GPU acceleration. In its current state, 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.

3.3.2 Micobenchmark

We aim to find out which part of AINT is the slowest. Knowing the bottleneck help us improving the performance. We aim to figure out 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 4. The microbenchmark was done using cProfile². And we visualized the data by converting cProfile traces to kCacheGrind format³ and pyprof2calltree⁴. We did our best effort to find components that spend the most time.

²<https://docs.python.org/3.2/library/profile.html>

³<http://kcachegrind.sourceforge.net/html/Home.html>

⁴<https://pypi.org/project/pyprof2calltree/>

| 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. | ✓ |
| 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 1: Tampering Detection

| 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 2: Variation Tolerance

| | 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 3: Seconds per frame for validating AINT pages

| 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 4: Microbenchmark

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.

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.

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

Table 5: Performance of AINT on 10 frames of 9 cells each and image hash as cache key.

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.

3.3.3 Caching

Cache saves repetitive computation, but the exact amount of saving depends on 1) user activeness 2) the length of the AINT session and 3) the design of the AINT web page. When the user does not move, all cells do not change and thus can benefit from cached validation results. Even if the user moves, only the cells that the cursor moves over and the cells where the user inputs need to be re-validated; other cells stay static and can benefit from caching. The longer an AINT session goes and assuming the same level of user activity, a longer interval of inactiveness can benefit from caching. And lastly, if the AINT web page is fine-grained tabularized, that means user interaction only affects a smaller number of cells, and thus a larger number of cells will be static and thus benefit from the cache. Theoretically, caching can reduce the AINT to only validate the integrity of the first frame for any arbitrarily long session given that there is no user activity. We did a simple test on 10 frames of The Example on CPU only. We list the average seconds per frame in Table 5. We emphasize that 1.32 second per frame is only the time for 10 frames, about 2.5s of user interaction if sampling at 250ms. We point out that there is no upper bound on the benefit of the cache. And thus our evaluation is for illustration only. We note that how cache key is computed affects the number of computations that we have to do, we will evaluate image hash and cryptographic hash as cache key generation method in the next section.

3.4 Security Guarantees of Image Hashes

Our hypothesis for image hash is that, in different rendering environments, it should return similar hash values for the *same content* and should return different values (larger than a threshold) for *different content*. In this section, we aim to find out if it is true.

3.4.1 Similarity Tolerance

In this experiment, we designed three test cases of similar-looking images to simulate the appearance on a web page, as shown in Table 6. 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 we used in AINT. 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 7. For image hashes, we indicate the hamming distance between two hashes. wHash 8 was used

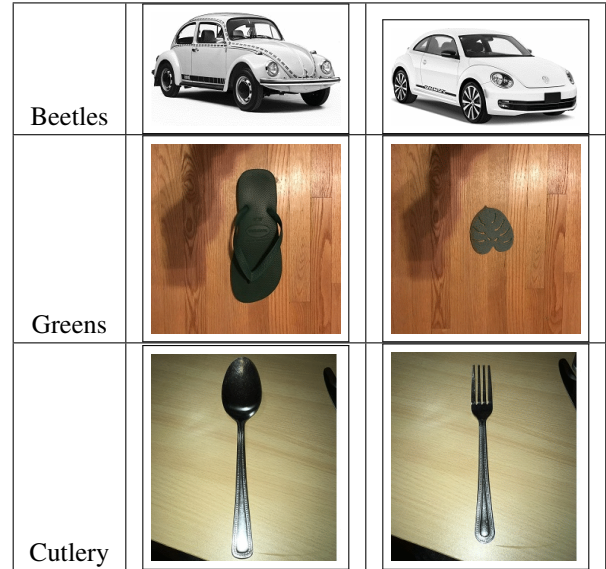


Table 6: Experiments used to test image hash on similar images.

| | Beetles | Greens | Cutlery |
|----------|---------|--------|---------|
| wHash 8 | 39 | 25 | 7 |
| wHash 16 | 104 | 96 | 36 |
| pHash 8 | 28 | 27 | 12 |
| pHash 16 | 160 | 118 | 117 |

Table 7: The hamming distance between the images in Table 6. The number after the hash function represents the output size in bytes.

| | wHash 8 | wHash 16 |
|--|---------|----------|
| 77 pairs of images in 16 unique hashes | | 0 |
| | pHash 8 | pHash 16 |
| | 0 | 0 |

Table 8: Number of pair-wise collisions of various hash function on randomly generated images.

by AINT with a threshold of 30, and only beetles' hamming distance is larger 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) use a greater wHash output size allows a hash value to capture more about the details of the image, and thus, when comparing two similar-looking images, the difference will be larger. This trend can be justified by the increasing difference between 8 bytes output and 16 bytes output. 2) use 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. 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.

| Output hash size | 8 bytes (64 bits) | 16 bytes (128 bits) | 32 bytes (256 bits) |
|------------------|-------------------|---------------------|---------------------|
| wHash | 3.62 | 3.66 | 3.95 |
| pHash | 0.94 | 1.15 | 1.81 |
| MD5 | n/a | 0.275 | n/a |
| SHA-256 | n/a | n/a | 0.43 |

Table 9: Average performance of hash functions on Caltech 101 in seconds

3.4.2 Collision Rate

A high collision rate on random images entails the 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 8. 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 the cryptographic hashes, there is no hash collision.

We show some examples of random images hashed to the same hash value in Table 11, and we detail some of the limitations of current image hash function that AINT uses. We observed the following: 1) different dimensions of the images do not prevent collision, as shown by the conugar face and motorbike example in Table 11. Images with different dimension look perceptually distinct to humans, but not image hashes. We suspect that it is due to the resizing operation of hash functions. Specifically, all input images are resized to have identical width and height. And because of this, long rectangular shaped input images suffers much more comparing to square shaped images, because the former must be squeezed causing information loss. 2) the shape of content contributes to the collision, as shown by the image with random background in Table 11. Due to its circular shape, its hash collides with many other images with circular shape. We suspect that it is also due to resizing. After resizing to 8 by 8, the details of the content is lost but the circular shape remains. 3) Since image hash removes color, the random background image collides with the hedgehog despite their color differences.

3.4.3 Image Hash Performance

As shown in prior work [3], Wavelet hash (wHash) and Perceptual (pHash) hash give good anti-tampering ability. We choose these two as our target hash functions for performance evaluation. We evaluate the performance of the two across different output sizes, from 8 bytes to 32 bytes. We illustrate the total time and average time in Table 9. We add the performance of cryptographic hash in the same table for reference. MD5 and SHA-256 are taken from HashLib in python. We tried evaluating the performance of MD5 and SHA-256 using OpenSSL because OpenSSL utilizes the Intel SHA extensions for hardware acceleration. However, the results are slower than HashLib in Python. The performance results suggest the use of pHash over wHash as the former is faster in every output size configuration.

4 Conclusion and Future Work

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

Image Hash and Threshold. As shown in Section 3.4, 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 trusted 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 [18]. 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 intention is collected. These 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 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 or the text behind it and OCR often gives inaccurate results. One solution is to leverage the GPU’s help. Since current OSs use hardware cursors, where GPU overlays the cursor frame buffer over another frame buffer for the rest of the display. It is possible for AINT to obtain two video recordings, one with the cursor and one without. The validation will be done without the cursor while checking for user initiation will be done on the version with the cursor. Running AINT on Android will not have the same problem because of the use of the touchscreen.

References

- [1] 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.
- [2] 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.
- [3] Content Blockchain. Testing different image hash functions, Dec 2018.
- [4] Graham Cluley. Viral clickjacking ‘like’ worm hits facebook users, 2010.
- [5] 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.
- [6] 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.
- [7] Facebook. Like button for ios, 2018-09.
- [8] 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.
- [9] Giorgio. Hello clearclick, goodbye clickjacking!, 2008.
- [10] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvania 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.
- [11] HTML5. Opera whole-page click hijacking via css, 2011.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] Krzysztof Kotowicz. Filejacking: How to make a file server from your browser (with html5 of course), 2011.
- [16] Eric Lawrence. Internet explorer 8 security part vii: Clickjacking defenses, 2009.
- [17] 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.
- [18] Dongtao Liu, Eduardo Cuervo, Valentin Pistol, Ryan Scudellari, and Landon P Cox. Screenpass: Secure password entry on touch-screen devices. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 291–304. ACM, 2013.
- [19] 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.
- [20] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Sava-gonkar. 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.
- [21] Marcus Niemi. Ui redressing: Attacks and countermeasures revisited. in *CONFidence*, 2011, 2011.
- [22] Marcus Niemi and Jörg Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.
- [23] 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.
- [24] Jeremiah Grossman (WhiteHat Security) Robert Hansen (SecTheory). Cursorjacking again, Sept 2008.
- [25] 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.
- [26] Tentacolo Viola. Cookiejacking, 2011.
- [27] 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.

A Tabularized Web Pages



Email transfer from your TD account

| | |
|---|--------|
| From Account | From |
| Amount | Amount |
| Recipient Email | Email |
| <input type="button" value="Transfer"/> | |

Figure 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 |
| <input type="button" value="Transfer"/> | |

Figure 2: The Example: AINT's interpretation

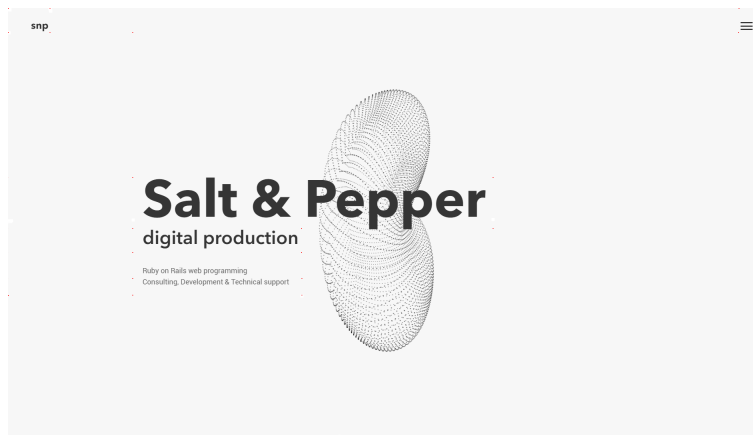


Figure 3: Salt & Pepper: user's view of a tabularized web page

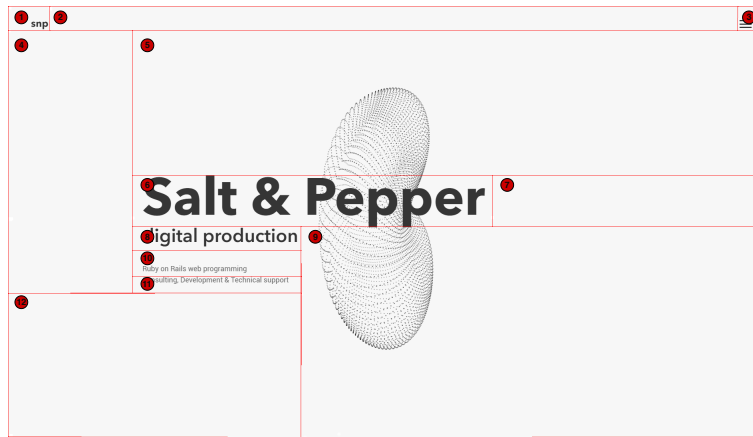


Figure 4: Salt & Pepper: AINT attempts to reconstruct the grid of a tabularized web page

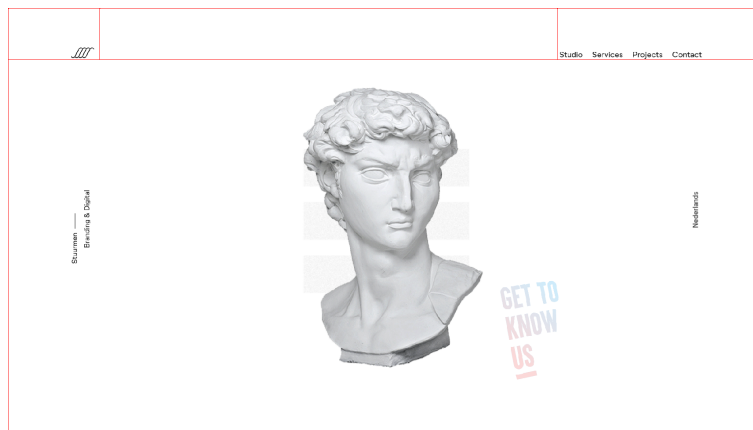


Figure 5: Unhappy: coarse-grained tabularization



Figure 6: Unhappy: fine-grained tabularization

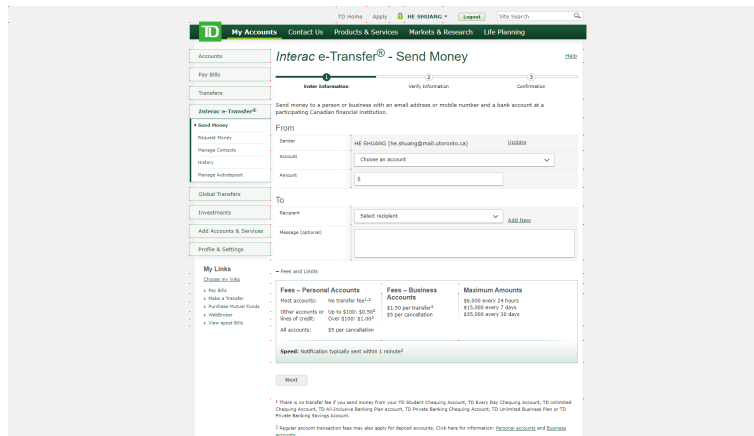


Figure 7: TD: an example of tabularized web page, user's view

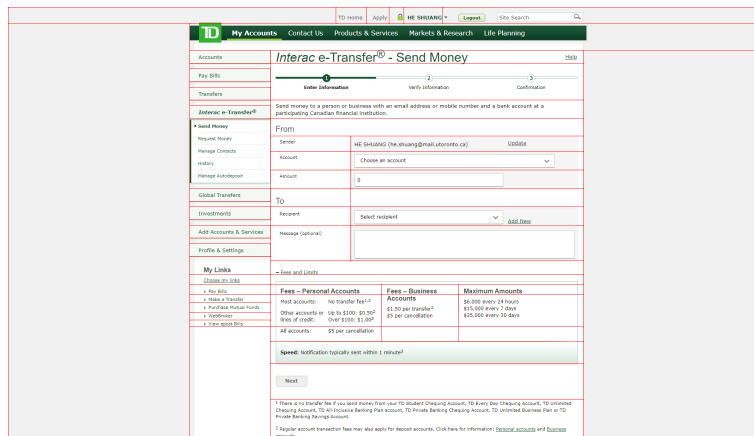


Figure 8: TD: AINT's interpretation

B Currently Supported Cursors


















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

Table 10: Currently Supported Cursors in AINT

C Random Images Evaluated to the Same Hash in Caltech 101

| Image | Image |
|--|--|
| <p>ketch</p>  | <p>Joshua Tree</p>  |
| <p>Cougar Face</p>  | <p>Motorbikes</p>  |
| <p>Random Background</p>  | <p>Watch</p>  |
| <p>Random Background</p>  | <p>Umbrella</p>  |

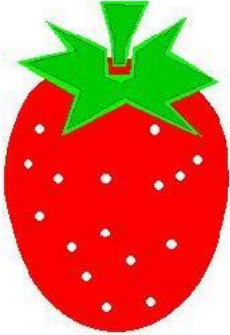






| | |
|--|---|
| <p>Random Background</p>  | <p>Strawberry</p>  |
| <p>Random Background</p>  | <p>Pizza</p>  |
| <p>Random Background</p>  | <p>Stop Sign</p>  |
| <p>Random Background</p>  | <p>Sunflower</p>  |
| <p>Random Background</p>  | <p>Hedgehog</p>  |

Table 11: Each row represents a pair of random images from different categories.

