Automated detection of security sensitive UI elements for Clickshield

Cristian Di Iorio, *1983177, Sapienza University of Rome*

Abstract—Clickshield is a tool aimed at protecting Android apps from clickjacking attacks. However it needs each security sensitive UI element it protects to be manually tagged. In this paper we will show how to implement an automatic system to detect and tag security sensitive UI elements in three steps. First we will analyze application code using a code-level analyzer that produces a particular kind of event graphs; then we will feed those graphs to Graph Neural Networks to detect whether they are security sensitive or not. Finally we will show how to tag those sensitive elements. This will greatly improve the usability of Clickshield.

Index Terms—UI attack, Clickjacking, Clickshield, UI-CTX, GNN

1 Introduction

Everybody owns a smartphone. In the wide plethora of possible attacks against them, some focus on the mobile user-interface (UI). An interesting type is *clickjacking*, which happen when attackers create an overlay that completely covers a security-sensitive app. Thus, while users think they are interacting with an innocuous overlay, they might be misled into performing unwanted actions on the app that is being covered.

To tackle this issue, [8] introduces ClickShield, a defense mechanism that works better than pre-existing ones like obscured flag and hide overlays. It works thanks to a novel image analysis technique, deblending. Clickshield focuses on the net effect of overlays, to try to interpret what the user is actually seeing. It manages to detect all malicious cases present in the ClickBench benchmark while still recognising the benign ones.

However, Clickshield only protects UI elements that have been manually flagged by developers as sensitive.

This is a glaring weakness and the authors themselves commented on the need for an automated tagging system of sensitive components. In this report we will implement such a system, using this three step approach:

1) first scan the application (in `.apk` format) using UI-CTX [4], a code-based UI analyzer that performs automated analysis of code to describe the behaviour of an app's UI elements through UI Handler Graphs (UHGs). These graphs model in a compact way the core intentions of a UI widget and they are a more detailed representation than raw code.
2) use Graph Neural Networks to solve a graph-level binary classification problem that assigns to each UHG a label indicating whether it is security-sensitive or not,
3) tag the UI elements that have a sensitive UHG so that Clickshield can protect them.

Manuscript delivered on June 29th 2025

All three steps are performed beforehand on a computer. At the end of the process we just described, we will be left with an application that hopefully has all its security sensitive elements tagged, so that Clickshield can recognise and protect them.

While GNNs have already been used in cybersecurity to analyze code graphs [9] [7], the core innovation brought by this report is combining them with UHGs, which are more specific, event-centered graphs that represent the code triggered by a UI widget.

The main focus in existing research has been to study functions on their own [9] while UHGs depict the whole chain of function calls linked to a single UI element. To the best of our knowledge, this three step approach to automated detection and tagging of sensitive UI elements has never been tried before.

2 Background

2.1 Clickjacking

A clickjacking attack is a UI attack where the attacker creates an opaque overlay that covers an app. While the user believes the overlay is benign, what's actually happening is that it tries to deceive the user into interacting with UI elements of the app that's being covered. It's easy to see that if the overlay covers something sensitive, the user could be misled into performing unwanted and dangerous actions like installing an arbitrary app or disabling security checks.

The security mechanisms implemented by Google, namely obscured flag and hide overlays, are either too simplicistic or too excessive.

The former works by tagging sensitive UI elements manually, then at runtime the mechanism can detect if they are covered by an overlay when clicked; however it has been proved to be ineffective for some attacks like Cloak & Dagger [3].

The latter works by removing all overlays; this is obviously a crude approach that creates backwards compatibility issues with some apps, especially ones whose core function is to create full screen overlays like screen filter apps [8].

2.2 Clickshield

Clickshield [8] is a defensive mechanism that offers protection from clickjacking attacks. To do so, it employs a novel image analysis technique, deblending. This technique involves using the raw pixel data of the display the user is seeing and the display without any overlays; those values are used to compute an opaqueness value and a uniformity score for the overlay layer. By checking those scores against a threshold, it can decide whether to accept an overlay or not. The only case in which an overlay is accepted is if it's semi-transparent and uniform.

This process is only executed when the user interacts with developer-flagged UI elements which are sensitive. The need for manual tagging of UI elements is an obvious Achilles' heel for this system, as stated by the authors.

2.3 UI-CTX

UI-CTX [4] is a static, code-based analysis framework designed to represent the true intentions of a UI widget (i.e. an UI element) of an Android app by linking the visible interface to the exact parts of code it triggers. It provides better accuracy compared to appearance-based and permission-based analyzers. It works in three phases:

First, it performs *Multi-Layer Knowledge Extraction*, meaning it parses the code to gather all informations relative to the UI layer. Then it performs static code analysis, such as inter and intra process control flow and data flow, to find each UI widget's real callback code. It does so by tracing backwards from event-registration calls to the exact sites (`findViewById` and `setContentView`) that link a UI element to its handler. Then branches that the widget can't participate in are pruned.

The second phase consists of *UHG construction*, when a UI Handler Graph (UHG) is built for each widget-event pair, where nodes are individual function calls and edges are their invocation relationships; it's important to note that node features contain the implementation of each function at the Dalvik-opcode level, instead of a simple name. Also, external libraries are summarized to make the graph more concise.

Finally, in the *Behavior Investigation* phase, it embeds each UHG into a fixed-length vector. Then these vectors can be used for further analysis. For example the authors use Agglomerative Hierarchical Clustering Analysis, an unsupervised algorithm that recursively merges and splits clusters based on distance. The end result is that vectors (and the relative UI elements that they represent) are clustered together based on functionality.

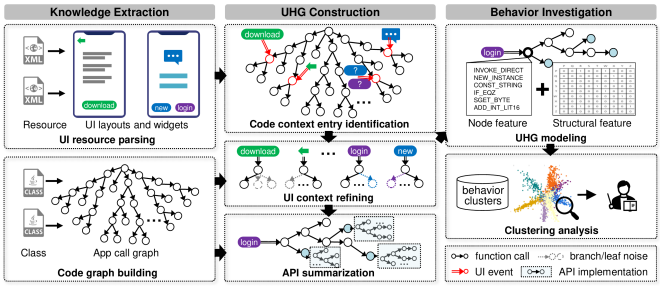

Fig. 1. Source: [4]

2.4 Graph Neural Networks

A Graph Neural Network [10] is basically a neural network that has been adapted to work on graph-structured data to perform normal machine learning tasks, like classification, regression or clustering [7].

The idea is that nodes of a graph are objects and edges are their relationships. Each node has an attached vector, its state, which is used by the two powerful components of GNN, the `aggregate` and `update` functions. Over several iterations, each node's state is iteratively updated by receiving information from its neighborhood and aggregating it. Then the output is produced using an output function. They can be used for different tasks of: node-level classification, edge-level classification and graph-level classification.

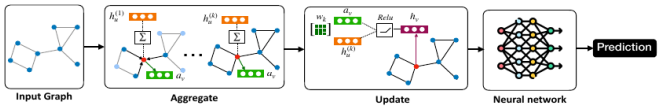

Fig. 2. Source: [7]

There are different GNN frameworks, where the main difference comes from the choice of the aggregate and update functions. Here are some of them:

- Graph Convolutional Networks (GCN), which use the convolution function on a graph's data to perform semi-supervised classification tasks. They provide a simple approach, however it comes at the cost of high computational overhead [7].
- Gated Graph Recurrent Networks (GGNN), which diverge from GNN for the use of Gated Recurrent Units, which remove the need to constrain parameters to ensure convergence [5]. They allow for a deeper analysis of graphs than GCN [12].

3 Overview of your proposed approach

Clickshield protects apps from clickjacking attacks. However, as we already mentioned before, Clickshield relies on developers manually recognising and tagging security sensitive UI elements. It's obviously impossible to do this manual analysis for all existing applications. So we propose an automated system that, starting from an app's `.apk` file, recognises and tags sensitive UI elements. To better explain our approach we will be following the steps of the pipeline described in the introduction.

3.1 Step 1: UI-CTX

We will use UI-CTX to study the behaviour of an application's components. It's the best choice for code-level analysis of UI widgets because it's resistant to code obfuscation techniques as it does not rely on class or method names to represent semantic information of function calls like some of the alternatives do [4].

At the end of the analysis, which works just like we described before, we are left with UHGs, which are the core representation of a single widget's behaviour for one event. So for each widget event there is a UI handler graph where the widget is connected to an event handler followed by the code contexts that are reachable by the widget, where each method call is represented as a node and the function it executes is represented as a feature. Then, UI-CTX also iteratively aggregates the semantic information of downstream graph nodes for each API call in widget code contexts, in an effort to summarise information and reduce the graph's complexity.

Then a lightweight graph embedding approach is used, which is efficient and retains sufficient code context. This decision is motivated by the fact that the task performed in UI-CTX is a broad clustering task of UI widgets based on behavior using HCA, which produces good results and even separates widgets of the same category based on how they work:

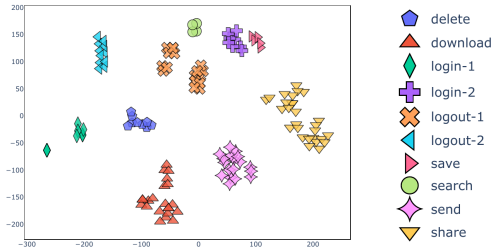

Fig. 3. Source: [4]

Here is where our approach differs from the one described in UI-CTX. In that paper the authors perform a task that is not specific, which is why HCA, an unsupervised ML technique, is used. Instead of HCA, we propose using supervised techniques such as Graph Neural Networks for our specialised use case.

3.2 Step 2: Graph Neural Networks

In the words of the authors of UI-CTX themselves: *"We can expect that if a high-quality labeled dataset for a specific task is available, leveraging learning models that are powerful in distilling correlations between data and their labels could lead to better performance on specific analysis tasks."* [4]

This is a perfect fit for our approach, since we have a *specific analysis task*, which is finding security sensitive UI widgets starting from their UHGs.

In other research, different kinds of graphs are used, such as control flow graphs, data flow graphs and abstract syntax trees [12]. Other research focused on combining those together [9], which are somewhat similar to UHGs, being a sort of *custom* graph. However in that paper Convolutional Neural Networks adapted for graphs were used, motivated by the fact that GNN need higher-level semantic information to work.

We suggest using GNN in our scenario because in our case we do have higher-level semantic information thanks to UHGs. Thanks to API summarization performed by UI-CTX and thanks to the use of Dalvik opcodes in UHGs, we have a precise semantic representation of operations performed by functions, which is more meaningful than simple method names.

As for the kind of GNN we are going to use, like we explained in chapter 2.4 there are multiple kinds. Our task is a binary graph-level classification [7]. For our use case we suggest using a Deep Graph Convolutional network, like in [13] or a Gated Graph Recurrent Network similar to [12]. Of course both would need to be adapted to our use case. The choice between models would largely be influenced by performance, computing power and dataset size. In particular, a deep graph convolutional network would require a significantly large labeled dataset of UHGs and it would also require considerable computing resources. Instead a Gated Graph Recurrent Network inspired by [12], would need a smaller labeled dataset and it would use less resources, due to having less layers than a DGCN. It may be better to prioritize a Gated Graph Recurrent Network based implementation, because as it will be explained in chapter 4, we will need to create the labeled dataset ourselves and that implementation needs less labeled data.

3.3 Step 3: Tagging

Step two labels UHGs as either security sensitive or not. So it is easy to create a list containing the widget IDs of security sensitive UI elements by looking at their UHGs.

Now we need to tag them in the code so that they can be recognized and protected by Clickshield. Existing Android defenses [1] rely on setting `View` attributes in the `Java` app code:

```
View.setFilterTouchesWhenObscured(true)
```

or in the `.xml` layout file:

```
android:filterTouchesWhenObscured="true"
```

We could use a similar approach by creating a new XML attribute, for example:

```
app:clickShieldProtected="true"
```

4 Evaluation

Graph Neural Networks need a labeled dataset to work, which means that we need to build a labeled dataset of UHGs that are security sensitive and some that are not. We suggest using as a starting point the dataset [11] produced by the authors of the paper, who scanned more around forty thousand apps from different market sources. Obviously

we will need to exclude some samples.

To have a rough estimate of how large our labeled dataset should be, we note that in UI-CTX, the authors report that the average UHG has around 1367 nodes and 11444 edges. This is just a bit bigger than the size of the graphs in the D&D dataset used in [13], which means that they are somewhat comparable. That dataset contains 1178 graphs of which 691 are enzymes and the rest are non-enzymes.

This means that at the very least to create our labeled dataset of UHGs we would need to manually label a minimum of 1000 UHGs as either security sensitive or not. The dataset also needs to be balanced, to avoid unwanted bias towards the eventual majority.

After building the dataset, the main metrics to evaluate the performance of our GNN should be:

- Receiver Operating Characteristic (ROC) with Area Under ROC (AUROC); the ROC curve plots the true positive rate against the false positive rate for each threshold.
- Precision-Recall (PR) curve with Area Under PR (AUPR); the PR curve plots precision against recall at each threshold.

We chose those since they are the most representative for our graph-level binary-classification task. We can also use precision and F1 score as secondary metrics.

As for computing performance, due to our use of complex GNNs and large datasets we will probably need a powerful computer like the one used in [4].

5 Related work

Computer vision represents an interesting alternative to Clickshield for clickjacking protection. Some research has already been carried on detecting *dark patterns* (deceptive UI designs) in mobile applications using a combination of computer vision and natural language processing techniques [6] [2]. However this work is only partially related to clickjacking and there is no certainty that it might be applicable to our scenario.

We have already cited some papers that employ more classical graphs (data-flow, control flow) compared to the UHGs produced by UI-CTX. However we chose to use the UHG representation since they are event-driven so they are a perfect fit for our use case. However UI-CTX is not perfect by any means, because it may not accurately capture widget functionalities if the application code is encrypted [4].

Some alternatives to our Graph Neural Network based approach are Convolutional Neural Networks adapted to graphs, like the one used by [9] but we have already explained in chapter 3 why we think our approach is better.

6 Conclusions

In this report we have shown that, by feeding to a GNN the graphs produced by code-level analysis of an app (UI-CTX), we can automatically detect security sensitive UI elements of an Android application. This is needed to enhance Clickshield [8], since it relies on manual detection of sensitive UI elements.

Future efforts should focus on implementing the proposed system and creating the labeled dataset described in chapter 4. Other research should also be focused on improving and updating Clickshield itself.

References

[1] Android Developers, "Tapjacking," https://developer.android.com/privacy-and-security/risks/tapjacking, Google LLC, Sep. 2024, accessed: 2025-06-12.

[2] J. Chen, J. Sun, S. Feng, Z. Xing, Q. Lu, X. Xu, and C. Chen, "Unveiling the tricks: Automated detection of dark patterns in mobile applications," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3586183.3606783

[3] Y. Fratantonio, C. Qian, P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the ui feedback loop," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017, pp. 1041–1057. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Fratantonio-Cloak-And-Dagger-From-Two-Permissions-To-Complete-Control-Of-The-UI-Feedback-Loop-wp.pdf

[4] J. Li, J. Liu, J. Mao, J. Zeng, and Z. Liang, "Ui-ctx: Understanding ui behaviors with code contexts for mobile applications," in *Network and Distributed System Security (NDSS) Symposium 2025*, February 2025. [Online]. Available: https://dx.doi.org/10.14722/ndss.2025.240238

[5] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1511.05493

[6] S. M. H. Mansur, S. Salma, D. Awofisayo, and K. Moran, "Aidui: Toward automated recognition of dark patterns in user interfaces," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 19581970. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00166

[7] S. Mitra, T. Chakraborty, S. Neupane, A. Piplai, and S. Mittal, "Use of graph neural networks in aiding defensive cyber operations," 2024. [Online]. Available: https://arxiv.org/abs/2401.05680

[8] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio, "Clickshield: Are you hiding something? towards eradicating clickjacking on android," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 11201136. [Online]. Available: https://doi.org/10.1145/3243734.3243785

[9] A. S. Saimbhi, "Enhancing software vulnerability detection using code property graphs and convolutional neural networks," in *2025 International Conference on Computational, Communication and Information Technology (ICCCIT)*. IEEE, Feb. 2025, p. 435440. [Online]. Available: http://dx.doi.org/10.1109/ICCCIT62592.2025.10928033

[10] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[11] UICTX, "Uictx demo data," 12 2024. [Online]. Available: https://figshare.com/articles/dataset/UICTX_Demo_Data/27266934

[12] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *CoRR*, vol. abs/1909.03496, 2019. [Online]. Available: http://arxiv.org/abs/1909.03496

[13] Y. Zhou, H. Huo, Z. Hou, and F. Bu, "A deep graph convolutional neural network architecture for graph classification," *PLOS ONE*, vol. 18, no. 3, pp. 1–31, 03 2023. [Online]. Available: https://doi.org/10.1371/journal.pone.0279604