AppBeach: A Static Behavior Checker for iOS Mobile Applications

Fang Yu, Steven Tai, Tang-Wei Shao and Wei-Ren Wang
Department of Management Information Systems
National Chengchi University, Taipei, Taiwan
yuf@nccu.edu.tw

Abstract.

AppBeach standing on App Behavior Checker is a new service to reconstruct and report behaviors of iOS mobile applications, using static binary analysis to reveal embedded functions within the executables. AppBeach adopts a distributed algorithm on call sequence counting via the hadoop framework, achieving a scalable static syntax analysis on executables of modern apps. The main idea is syntactically counting call sequences that are embedded in iOS executable. This is done by distributing routines to mappers with the assembly tool that resolves explicit and implicit system method calls that are embedded in the iOS executables. The reducer then collects the counting from mappers to characterize the behaviors of apps. We learn patterns of malicious behaviors from the difference of pairs of normal and malicious apps, and report the probability of potential behaviors of commercial apps by matching these patterns to their call sequence counts.

1. Introduction

As mobile devices and their applications become increasingly popular, security of mobile applications becomes the stunning block; users have to face threats of improper uses of their private information that can result in malicious attacks [15]. It has been shown that solely relying on license agreements has limited protection. One famous example is the social app Path [16] that retrieves and transmits iOS users address books to external devices without any notification. In addition to malicious apps, many suspicious behaviors are actually embedded in the third part library that are invoked neither with developer's intention nor with user's awareness. It is hence essential to develop a systematic approach to analyze mobile applications to discover potential behaviors of these apps, not only from the descriptions but from the applications themselves.

We have proposed in [12] a static binary analysis on iOS executables. We realize the techniques in this presented tool AppBeach [14], providing a systematic service to analyze and characterize mobile applications based on direct analysis on their executables. The analysis is particularly useful to detect sensitive behaviors of apps that are embedded in their executable but may or may not appear to users. Our main idea is resolving and counting system call (i.e., objc_msgSend) sequences within the assembly of the app executable. This information is particularly essential to iOS application users since many functions can be called without consent dialogs, i.e., runtime permission dialogs. Users may not be aware of their sensitive data have been accessed in many cases.

2. Related Work

To analyze mobile application binaries, both static and dynamic analysis techniques have been proposed [12][3][10][2]. Dynamic analysis primarily relies on observing (systematic) executions of the binary on (predefined) sets of runs. Dynamic binary instrumentation frameworks such as Valgrind [8] and Pin [4] facilitate such tool development. Most of these techniques and tools work on x86 instructions (rather than Arm/Darwin instructions) and cannot be applied directly to iOS applications. Previous work on android mobile applications mainly adopts dynamic analysis, given that the android platform provides well simulation of the binaries. For instance, Barbic et al. [1] draw system call dependency graphs that trace program executions, log system calls, and trace how parameters propagate, and finally compute the control flow of their behaviors. Zhou et al. [13] propose permission-based behavior footprints to discover malicious behaviors that collect permissions requested by known malware graphs. However, there is no such simulation tool support to public iOS applications. Apple provides an emulator under the XCode develop environment that works for x86 executions. The simulation requires recompiled applications with source codes to be executed. That is to say, the simulation tool cannot be used to analyze apps (no source code available) that are directly downloaded from Apple App Store. Another challenge of dynamic analysis is to achieve high coverage of executions. Automatic test generation techniques have been proposed to address the issue.

Static analysis on the other hand provides a direct analysis on source codes or binaries without executions. Mann et al. [7] adopt static analysis to detect privacy leaks in Android applications. They identified private information sources such as user's location, contacts, calendar events, and network communications. They label the parameters with security levels. Variables associated with personal data are given high security levels to restrict unauthorized methods to access. Egele et al. [3] present PiOS, the first static binary analysis tool for detecting privacy leaks in iOS applications. Similar to our tool, PiOS decrypts and analyzes

binaries of iOS applications directly. They present set of techniques that can be used to build control flow graphs of system calls of the binaries. This is done by conducting data flow analysis on assembly codes with additional information in their head files. Based on the flow, they check whether applications contain suspicious flaws for privacy leaks. PiOS parses the structure of the assembly to discover instructions in the execution order, showing the feasibility of static binary analysis on iOS applications. Compared to PiOS, we adopt a light way analysis on counting system call sequences that are embedded in the executable in syntactic order. We trade off precision with performance. Werthmann et al. [11] present PSiOS on the other hand, using application framework to protect privacy by enforcing the behaviors of iOS applications under customized strategies. Yu et al. [12] propose static binary analysis to resolve system call functions that are invoked by the executable. The detection is based on system call counting, showing a light way static analysis with efficiency.

We adopt static binary analysis [12] with extension to call sequence counting in this work. Our static analysis takes advantage of soundness by analyzing source codes without actually executing them. We perform the analysis on the executable binaries to identify all system functions that have been invoked. The objective of this work is to present a new tool AppBeach that features an effective static approach to analyze behaviors of iOS apps. We adopt static binary analysis to count call sequence appearance from binaries. Unlike previous work on analyzing control flow graphs [3] and using predefined frames to enforce privacy policies [11], we characterize apps via the list of call sequence counts. Livshits and Jung [6] conduct binary analysis on finding proper placements of consent dialogs to improve privacy protection. We directly discover sensitive behaviors from binaries. We report our analysis online against thousands of popular mobile applications and show that the characterization on call sequence counts is sufficient to discover the behaviors embedded in executable in many cases.

3. AppBeach

The architecture of our tool AppBeach is shown in Figure 2. We first extract decrypted assembly of apps from jailbroken devices with mac tools. The step is similar to the techniques presented in [12]. We then dump all the methods and classes that are resolved by the assembly tool IDA Pro [5]. We then count appearances of classes and methods in a syntactic order as a syntax characterization of functions invoked by the executable. To achieve scalability, the counting is done in a distributed fashion via the Hadoop MapReduce framework. One extension of AppBeach [12] is counting call sequences instead of counting single calls (1-sequence in this context). Since sensitive behaviors of applications are actually carried out by executing a sequence of system calls, these calls must be invoked in the right order to

bring out the behaviors correctly. For instance, to upload user's GPS location to an external device, an app has to access the GPS location before sending the GPS locations outward the device. In other words, if the app sends a message outward before it accesses the location, the information sent could be null or invalid. Unlike PiOS that builds control flow graphs, we present a light static analysis on counting call sequence in a syntactic order.

To count n-sequence calls, the mapper accumulates the call sequence and does not write the appearance to dataset (that is later summarized by the reducer) until n calls have been collected. That is to say, the modified map routine keeps call sequences (instead of a single call) as keys. After that, the mapper reads and shifts one call at a time to add the next n-sequence. Below it shows part of the code segment of mapper to collect n-sequence call where n is equal to windowSize.

```
while (tokenizer.hasMoreTokens()) {
    sequence.add(tokenizer.nextToken());
    if(sequence.size == windowsSize){
        String addWord="";
        for(String w:sequence){
            addWord+=w;
        }
        word.set(addWord);
        output.collect(word, one);
        reporter.incrCounter(Counters.INPUT_WORDS, 1);
        sequence.removeAt(0);//remove first
    }
}
```

The reducer then counts call sequences by accumulating pairs with the same key. The result is the list of n-sequence call counts. For instance, an example of 3-sequence call counts to access user events consists of accessing the date (in NSDate class), calendar (in NSCalendar class) and event (in EKevent class). The count indicates the number of appearances of such call sequence in the executable.

NSDate EKReminder EKEvent: 2
EKReminder EKEvent NSDate : 4
EKEventStore NSCalendar NSDateComponents : 5

To characterize malicious behaviors, we build malicious applications that are developed in pairs; each pair consists of one normal app and its abnormal counterpart. Both have identical behaviors except an inserted malicious behavior that we would like to characterize. Figure 2 shows the steps to characterize malicious behaviors. Unlike simply counting method calls in [12], we count call sequences as characterization of app behaviors.

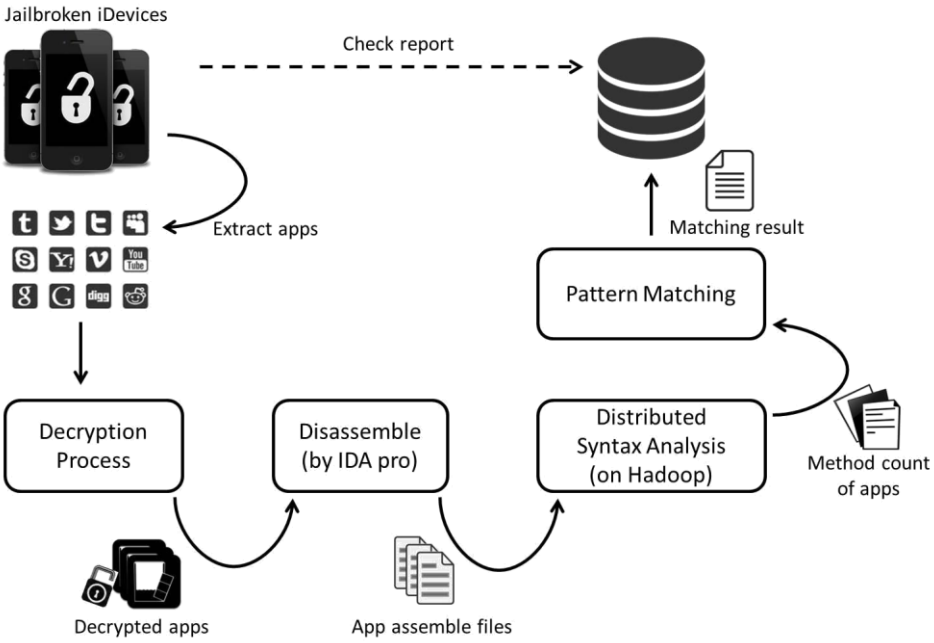

Fig. 1: The system architecture of AppBeach

We insert the malicious behavior that we target to characterize in the source code as the abnormal one, and leave the original code as the normal one. That is the pair are identical except the abnormal one has an embedded malicious behavior. After compiling their source codes of the pair applications, we apply the presented binary analysis on their executables and characterize the difference of their behaviors as the malicious signature for the embedded behavior.

Since we use the call sequences analysis we proposed to generate the pattern of sensitive behaviors, under different given sampling condition, we generated different pattern for same behaviors, for example, we conclude the call sequence for both of class invocation and method invocations, this will generate different patterns, on the other hand, the length of sampling sequence also brings out different patterns. The example shown in Table 1 gives the different pattern for the same behavior on accessing location.

We build a pattern library as the collections of these malicious signatures and use them later to detect whether the target app includes the malicious signature. If so, the target app may be able to execute the malicious behavior; cannot execute the malicious behavior,

otherwise. It is a sound approach by counting with respect to the malicious signature. Note that the learned malware signature depends only on the embedded malicious behavior but not the application itself.

Table 1: Different pattern for accessing location on sampling variation
(both considered the invocation on methods but classes)

1-sequence
setDelegate 1
setDesiredAccuracy 1
startUpdatingLocation 2
2-sequence
DesiredAccuracy Delegate 1
didReceiveMemoryWarning startUpdatingLocation 2
setManager init 1
setDistanceFilter DesiredAccuracy 1
startUpdatingLocation setDistanceFilter 2

4. Evaluation

We have realized our idea in the tool AppBeach that stands on App Behavior Checker. AppBeach contributes the society on analyzing iOS apps in three folds: (1) tools that are publicly available for download, allowing examination of applications for public use; (2) patterns of sensitive behaviors on call sequence counts that are collected from difference of pairs of self-developed apps; and (3) the app database that reports the analysis results on analyzing commercial apps downloaded from App Store.

AppBeach provides several scripts on the website with detailed instructions for evaluations. These scripts can be used to reproduce the analysis results. First, it has a decryption script to decrypt iOS executable downloaded from Apple App Store automatically. Note that executables that are downloaded from Apple App Store have parts of machine instructions are encrypted and are unable to be analyzed directly. Decryption is the first and necessary step to analyze online apps. AppBeach then runs a python script tracker that

resolves and counts system calls that have been invoked in the executable. This python code has to be run with the disassembler IDA Pro. The python code counts one call sequence of system calls. Third, AppBeach provides the python script to compare call counts of apps to patterns, finding whether the target sensitive behaviors could perform by the executables.

We have built several patterns of sensitive behaviors of apps. These patterns are specified as sequence counts of classes and methods. Two major categories are (1) retrieving sensitive information, including the user information (e.g. Address book, Calendar) and device information (e.g. GPS Location), (2) conveying information outward the mobile devices (e.g. transmit data via HTTP or TCP). For each purpose there are several ways to implement, while the essentials are what the system method calls have been invoked in these implementations. We implement and insert these sensitive behaviors from built-in frameworks or public packages. A call sequence pattern (shown in Figure 3) is the difference of the normal self-developed app and the modified version with the sensitive behavior inserted. The pattern page shows the behaviors we have collected and used in our examination. Each pattern is represented as method (or class) sequences and their counts. These patterns can be further extended to other behaviors, such as advertisements.

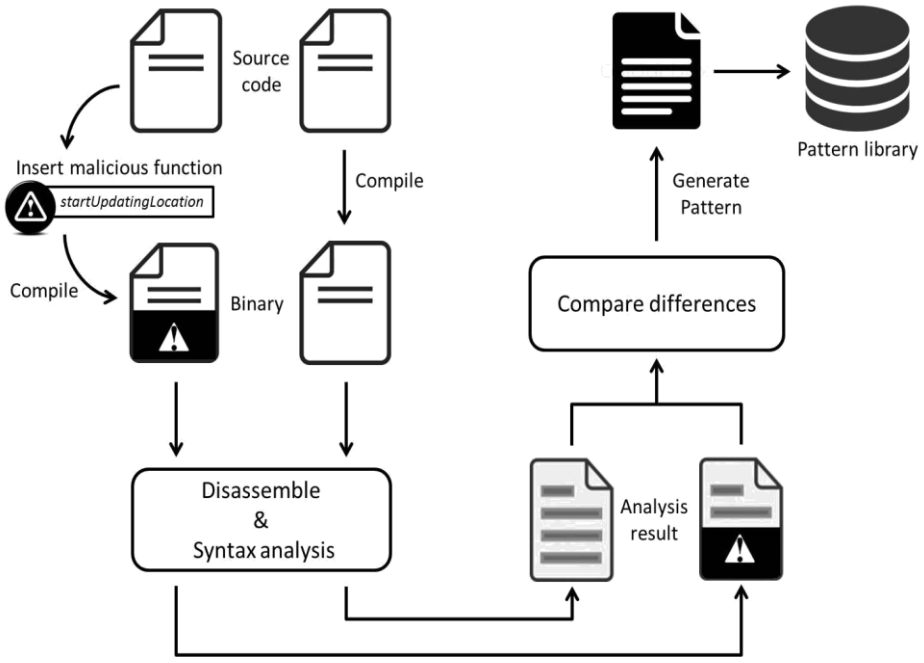

Fig. 2: The processes of building the malicious pattern library

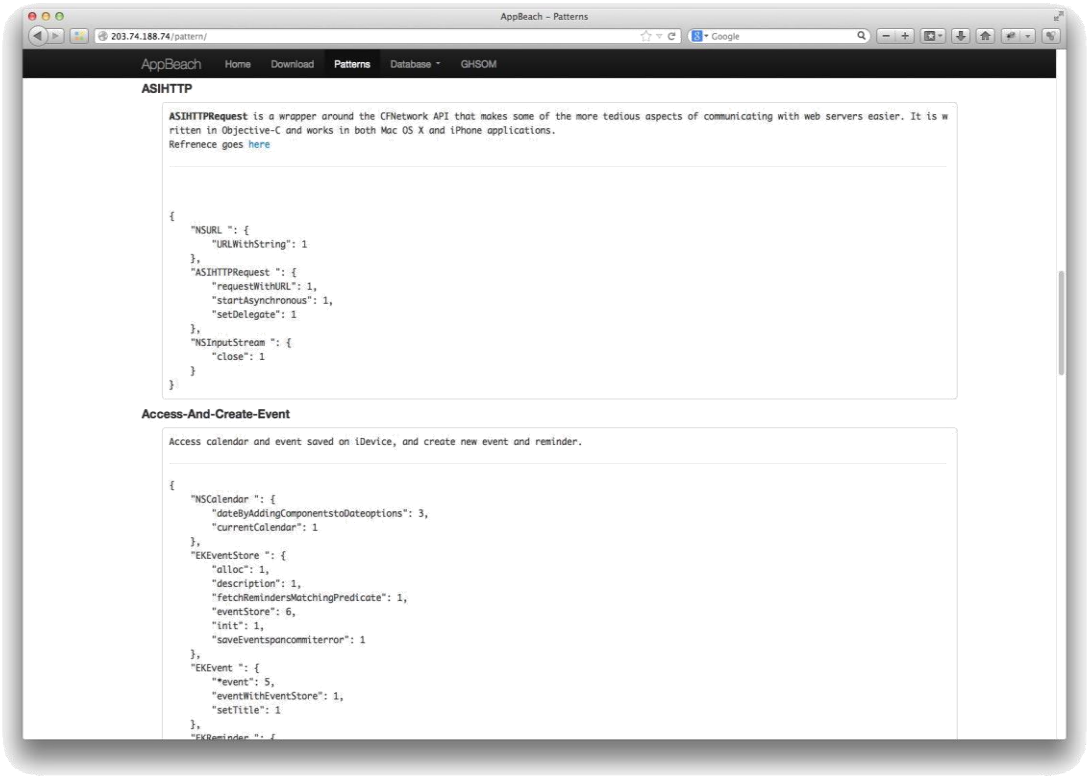

Fig. 3: Sampled Patterns of sensitive behaviors

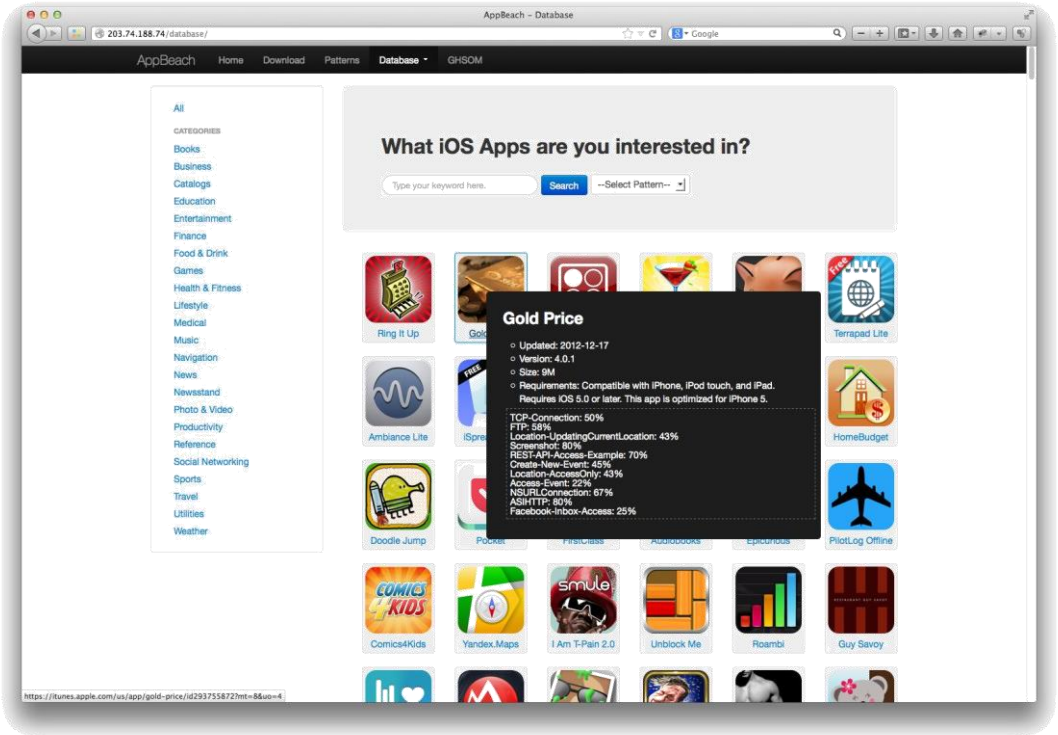

Fig. 4: The Sceenshot of App Behaviors in AppBeach

Finally, AppBeach provides the analysis database (shown in Figure 3) that reports our analysis results on most popular apps in each category listed in the Apple App Store. These 6000+ iOS applications are directly downloaded from Apple app store to be analyzed. We report the matched sensitive behaviors with probabilities: for each behavior pattern, an index 100% indicates that the app has its resolved call sequences containing all the call sequences specified in the pattern. In many cases, an app invokes only parts of call sequences specified by a pattern. For instance, an index 50% associated with an app for a specify behavior indicates that only half of the call sequences of the pattern are matched by the resolved sequences of the app.

5. Conclusion

We present the tool AppBeach that automates the process of extracting and decrypting iOS applications, and disassembling and resolving call invocations for characterizing behaviors of applications with call sequence counts. We build patterns on several sensitive behaviors and check whether these behaviors are embedded in thousands of popular mobile applications. The analysis results are public available in http://soslab.nccu.edu.tw/appbeach.

Acknowledgement

This work is partially funded by the grants MOST-103-2221-E-004-006-MY3 and NSC-102-2221-E-004-002-.

References

[1] D. Babi´c, D. Reynaud, and D. Song, "Malware analysis with tree automata inference," *In Proceedings of the 23rd International Conference on Computer Aided Verification, CAV'11*, pages 116–131, 2011.

[2] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," *Int. J. of Req. Eng*, 2001.

[3] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios, "Detecting privacy leaks in ios applications," In *NDSS*, 2011.

[4] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

[5] IDA Pro, https://www.hex-rays.com/products/ida.

[6] B. Livshits and J. Jung, "Automatic mediation of privacy-sensitive resource access in smartphone applications," In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 113–130, Berkeley, CA, USA, 2013. USENIX Association.

[7] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1457–1462, 2012.

[8] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[9] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for dynamic analysis of ios applications," In *Proceedings of the 2011 IFIP WG 11.4 International Conference on Open Problems in Network Security*, iNetSec'11, pages 65–77, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] H. Theiling, "Extracting safe and precise control flow from binaries," In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, RTCSA '00, pages 23–, Washington, DC, USA, 2000. IEEE Computer Society.

[11] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios, "Bring your own privacy security to ios devices," In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[12] F. Yu, Y.-C. Lee, S. Tai, and W.-S. Tang, "Appbeach: Characterizing app behaviors via static binary analysis," In *Proceedings of the 2013 IEEE Second International*

Conference on Mobile Services, MS '13, pages 86–93, Washington, DC, USA, 2013. IEEE Computer Society.

[13] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS'12, 2012.

[14] AppBeach, http://soslab.nccu.edu.tw/appbeach, 2014.

[15] Find and Call, http://www.wired.com/2012/07/first-ios-malware-found, 2012.

[16] Path, http://www.wired.com/2012/02/path-social-media-app-uploads-ios-address-books-to-its-servers/, 2012.

[Authors]

Dr. Fang Yu is an Associate Professor in the Department of Management Information Systems at National Chengchi University. He received his PhD degree and the 2010 outstanding dissertation award in Computer Science from University of California, Santa Barbara. His research interests broadly span software security, verification, formal methods, automata theory, and membrane and neural computing. He served the program committee of ACM LCTES, IEEE CloudCom, IEEE SCC, IEEE BigData, ATVA, etc. and co-chaired Infinity 2011. He is the current executive editor of International Journal of Information and Computer Security.

Steven Tai and Wei-Ren Wang are the master students in the Department of Management Information Systems at National Chengchi University. Tang-Wei Shao is the master student in the Department of Information Management at National Taiwan University. All of them are/were the members of Software Security Lab at National Chengchi Univesity.