BossPro: a biometrics-based obfuscation scheme for software protection

Torben Kuseler, Ihsan A. Lami, Hisham Al-Assam
Applied Computing Department
University of Buckingham
Hunter Street, Buckingham, MK18 1EG, UK
(phone: 44-1280-814080; email: first.last@buckingham.ac.uk)

ABSTRACT

This paper proposes to integrate biometric-based key generation into an obfuscated interpretation algorithm to protect authentication application software from illegitimate use or reverse-engineering. This is especially necessary for mCommerce because application programmes on mobile devices, such as Smartphones and Tablet-PCs are typically open for misuse by hackers. Therefore, the scheme proposed in this paper ensures that a correct interpretation / execution of the obfuscated program code of the authentication application requires a valid biometric generated key of the actual person to be authenticated, in real-time. Without this key, the real semantics of the program cannot be understood by an attacker even if he/she gains access to this application code. Furthermore, the security provided by this scheme can be a vital aspect in protecting any application running on mobile devices that are increasingly used to perform business/financial or other security related applications, but are easily lost or stolen. The scheme starts by creating a personalised copy of any application based on the biometric key generated during an enrolment process with the authenticator as well as a nuance created at the time of communication between the client and the authenticator. The obfuscated code is then shipped to the client's mobile devise and integrated with real-time biometric extracted data of the client to form the unlocking key during execution. The novelty of this scheme is achieved by the close binding of this application program to the biometric key of the client, thus making this application unusable for others. Trials and experimental results on biometric key generation, based on client's faces, and an implemented scheme prototype, based on the Android emulator, prove the concept and novelty of this proposed scheme.

Keywords: Biometrics, Mobile applications, Obfuscated interpretation, Software protection

1. INTRODUCTION

Software anti-piracy protection techniques attempt to contain/limit the financial damage of software piracy or misuse of any kind. Software misuse is rapidly increasing every year and it was quoted to cost the industry some $51.4 billion in 2009 [1]. One of the major concerns to software developing companies and software users in today's highly mobile and connected world is the distribution of "cracked", and the use of unlicensed software. On the other hand, mobile applications on Smartphones or Tablet-PCs, e.g. iPhone/Android based devices, are increasingly used to perform financial/business or security related transactions everywhere anytime. As these mobile devices are easily lost or stolen, the protection of data and applications on such devices against unauthorised access/use becomes even more important.

Biometric-based (or data regarding one's identity, or something you are) authentication, knowledge-based (or data of something you know, e.g. a password) authentication and object-based (or data about something you have, e.g. a token) authentication have been used extensively in various remote communication to validate a client to an authenticator [2]. In contrast to object-based or knowledge-based authentication factors, in biometric-based authentication a legitimate client does not have to carry or remember anything to perform the authentication. However, biometric authentication, which is known to be more reliable than the other traditional authentication methods, requires only the physical presence of the client, which makes it a very convenient and simple to use for authentication.

Current generations of personal computers, notebooks, Smartphones or Tablet-PCs feature a wide variety of sensors (e.g. camera, microphone or multi-touch displays) that can be easily employed to capture a client's biometrics.

This paper proposes to use a biometric key, generated from fresh and real-time captured client biometric data, in conjunction with obfuscated interpretation to protect the "execution of a software application" on the client's device.

Without presenting the correct biometric key to the system, the obfuscated program will not run at all or will produce an incorrect authentication for any illegitimate client.

A prototype of BossPro is implemented on an Android platform emulator. It is designed in such a way that it can handle both normal (unprotected) applications as well as obfuscated (protected) applications at the same time. Also, the proposed methods can be combined seamlessly and trouble-free with other software protection techniques, e.g. opaque predicates or lexical/control flow transformations as the instruction obfuscation is employed on the result of previous transformation steps. Additional software protection methods that increase the number of instructions can further enhance the security of the obfuscated interpretation.

The rest of the paper is organised as follows: Section 2 describes the background and outlines related work of software protection techniques and biometric-based key generation. Section 3 introduces the general concept of the proposed biometric-secured obfuscated interpretation scheme. Section 4 describes the implementation of the prototype application based on the Android emulator and discusses the trial and experimental results. Finally, we conclude the paper and outline future work in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Software protection techniques

Software protection can be broadly categorised into three main groups [3]: 1) Software watermarking, 2) Software tamper proofing or tamper resistance, and 3) Software obfuscation. Software watermarking adds visible or hidden information to source code to prevent software theft or to proof the original ownership, if a misuse of a piece of software has been discovered. To generate resilient, cheap and stealthy software watermarks various methods based on, for example, opaque predicates [4], register allocation [5] or self-validating branches [6] were proposed. Software tamper proofing or more precisely tamper resistance (as every protection mechanism can be bypassed with sufficient knowledge, time or resources) tries to prevent illegal modification or distribution of software. Examples of proposed tamper resistance techniques include self-modifying and self-decrypting Integrity Verification kernels (IVK) [7] or dynamic integrity checking [8]. Software obfuscation is similar to tamper resistance in that it also aims to protect the code against malicious modifications. However, in contrast to tamper resistance, software obfuscation [9] attempts to transform a piece of software into an equivalent program that has the same behaviour but is more difficult to reverse-engineer and therefore harder to manipulate by an attacker.

Neither watermarking nor tamper resistance or obfuscation techniques alone can guarantee a fully protected software [10]. But even though one technique on its own can be easily broken [11], a combination of several methods can “raise the bar” substantially. This makes any attempt to fool such protection system uneconomical for the attacker.

Monden has introduced a framework for obfuscated interpretation [12] where a hardware implemented finite state machine (FSM) was proposed. This ASM “retranslates” the instructions of an obfuscated program into the original ones during program execution/interpretation (Figure 1). The main security concept of obfuscation interpretation is that a program can be considered "secure" (in terms of reverse-engineering) if an attacker can not understand the real semantics of the program from the available code instructions, without having the state transition rules for the retranslation (reverse obfuscation). A major drawback of the framework proposed by Monden is that the change of the transition rules is very difficult in the hardware based implementation of the FSM during development and nearly impossible once the hardware unit is integrated into the end-user device. In addition, dummy instructions must be injected into the code to guarantee the correct interpretation by the FSM. However, dummy instructions could leak information about the real semantics of the program code and so make attacks possible. To overcome these problems, Zhang [13] proposed an obfuscated interpretation framework that uses a permutation-based interpreter (PMI) implemented in software. The PMI allows in their proposed framework an easy change of the transition rules and does not require any additional dummy instructions. Recently, Zeng [14] has developed a software watermarking scheme based on obfuscated interpretation.

BossPro uses a protection scheme similar to [13], but uses biometric generated keys to (de-)obfuscate instructions during program development and execution/interpretation. This will tightly bind the obfuscated software to the legitimate client and shall remove the requirement to hide the interpreter from the program user (or a possible attacker) as well as the necessary encryption of the permutation rules described by Zhang.

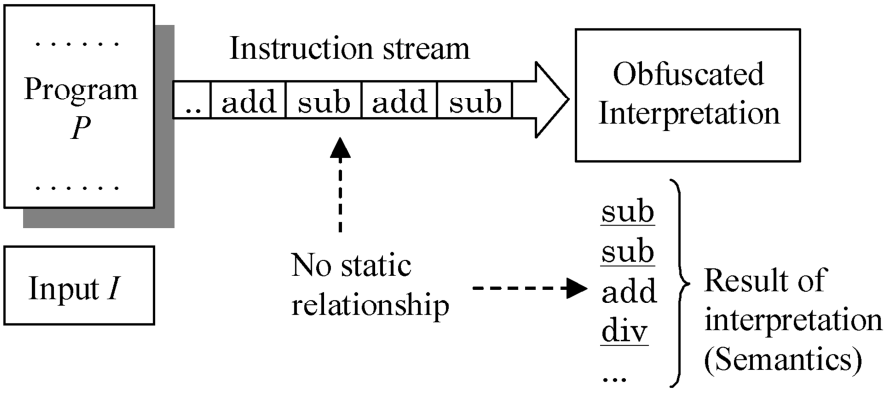


Figure 1. Concept of obfuscated interpretation as originally proposed in [12]

2.2 Biometric key generation methods

Biometric cryptosystems have been proposed to provide stronger security mechanisms by combining biometrics with cryptography. Biometric cryptosystems can be broadly categorised into three main approaches: (i) key release, (ii) key generation, and (iii) key binding.

In the key release approach, the cryptographic key and the biometric data of a client are stored as two separate identities at different hosts where the key is released only when an authentication attempt of the client is successful. This method is straightforward and easy to implement but it has two major drawbacks [15]. The first drawback is due to the fact that biometric templates are not secure and that the "biometric matcher" can be overridden. The second drawback is due to the fact that cryptographic keys are not secure because they are not combined with biometric data when compared to the other two approaches.

In the key generation approach, a cryptographic key is directly derived from the biometric data without storing it anywhere. Typically, biometric features in this approach are represented as a binary string and the robust bits are selected as a cryptographic key. It has been shown that such methods have high False Rejection Rates (FRRs) which them an impractical method [16].

Finally, in key binding approaches, the biometric template and the key are combined in a way that makes it computationally infeasible to retrieve the key without previous knowledge of client's biometric data. The cryptographic key is randomly generated during the enrolment stage. Then it is discarded after combining it with the biometric data. At the authentication stage, the cryptographic key is released only if the query biometric sample is matched. It is known that biometric data are fuzzy due to intra-class variations resulting from the differences between the freshly captured biometric sample and the enrolled templates. On the other hand, cryptographic keys have to be precise and repeatable every time. Therefore, there is a need to employ error correction techniques such as Error Correcting Codes (ECC) to bridge the gap between the fuzziness of biometric and the preciseness of cryptographic keys.

BossPro adopts the key binding approach for our implementation based on face biometric. Note that, throughout this paper "key binding" and "key generation" are used similarly. Figure 2 illustrates the process of biometric key binding. The cryptographic key is fed into an ECC encoder and then XORed with the binary representation of biometric data to produce a biometric lock or helper data. After that, the key is discarded and the biometric lock and the hash of the key are saved. The binary representation of the biometric data is calculated from extracted biometric features where a client-based transformation such as random projection [17] can be employed to produce cancellable biometric followed by a biometric binarisation. At the authentication stage (key retrieval stage), the binary representation is calculated using a fresh biometric sample in the same way as described above and then XORed with the biometric lock. Then, an error correcting technique is used in the decoding mode to tolerate intra-class variation. The decoding is successful and the key released if the difference between the reference biometric sample(s) and the fresh biometric sample is within a certain predefined threshold.

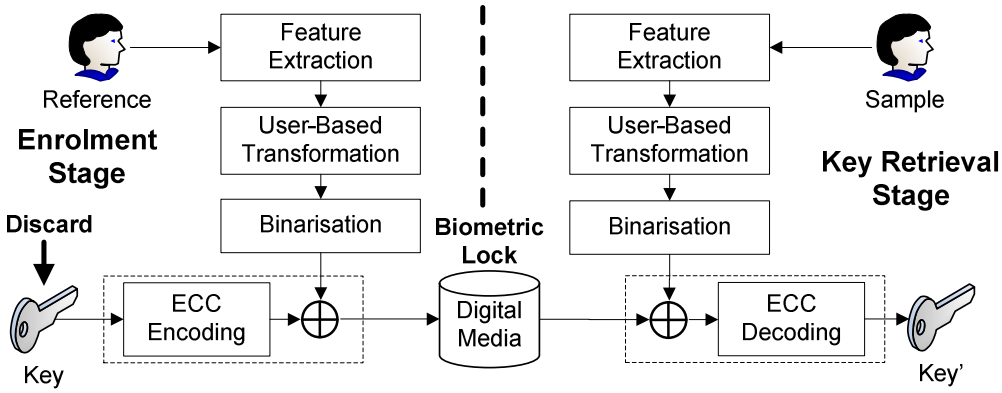


Figure 2. Biometric key binding process

3. BIOMETRIC-SECURED OBFUSCATED INTERPRETATION

The two base elements of the BossPro scheme are (1) the biometric key generation and (2) a standard software development cycle supplemented by an optional specification step that defines the application elements to be obfuscated (Figure 3).

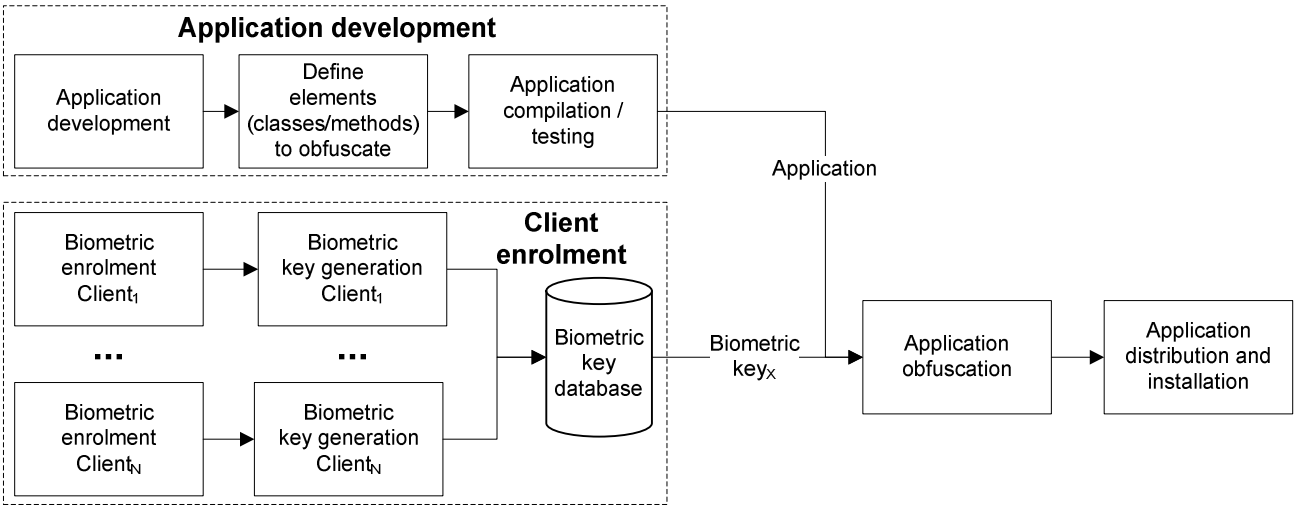


Figure 3. Application development and client enrolment

During an enrolment process, biometric keys are generated from the freshly taken client biometrics (e.g. face) and stored for later use (application obfuscation) in a database together with additional client information (e.g. unique client identifier). To protect the sensitive client-specific biometric data, only cancellable / revocable biometric keys [18] are used and stored in the database.

A possible usage scenario for BossPro is a remote authentication process between a client and a bank in a mCommerce transaction. For example, the bank wants to offer their services only to enrolled clients, who have been successfully verified to the bank at the moment of system use. Another application example is when any software developer or business company (e.g. bank) wants to distribute a new (or an updated version of) their biometric-secured software application to their enrolled clients. Then, a standard software development cycle (e.g. Java development for Android based mobile devices) is executed. If it is not possible or desired that the complete application is protected by obfuscated interpretation (e.g. when some parts may be used as libraries by other (not protected) applications), then an optional development step can specify the elements to be obfuscated. Otherwise the complete application code is obfuscated. If

the software works as expected, the new (or updated) application is obfuscated using the biometric key. In this process, the original program instructions are substituted (obfuscated) with instructions selected on the basis of each client's stored biometric key making the resultant application program code uniquely tailored/dependent on each client. This individual biometrically-secured obfuscated application "code" is then distributed to its associated and enrolled client for installation on the client's device, e.g. Smartphone. To enhance the authentication process, some transactions may encode a "nuance" generated at the communication/transaction time to secure the transmitted program code and so eliminate the replay attacks.

Secure storage of the client's cancellable biometric keys in the proposed scheme removes the necessity of re-enrolment of the client for each new (or updated) version of the application, which would be expensive, time-consuming and inefficient in scenarios with many applications and enrolled clients.

Once a client wants to execute a possible protected application on the mobile device, for example for authentication purpose for a financial transaction, the proposed scheme shall start the application and check if this application is protected and so needs an obfuscation interpretation (Figure 4). As the general layout and the instructions of an obfuscated program are not distinguishable from an unprotected program during interpretation/execution, an additional label is inserted during the obfuscation process. If this label is present in the application, or in some parts of the application, then the scheme invokes a process, which generates a new biometric key from a freshly taken client biometrics (e.g. face), and passes this key to the application interpreter to perform the obfuscated interpretation. Otherwise, the application is executed normally by the interpreter.

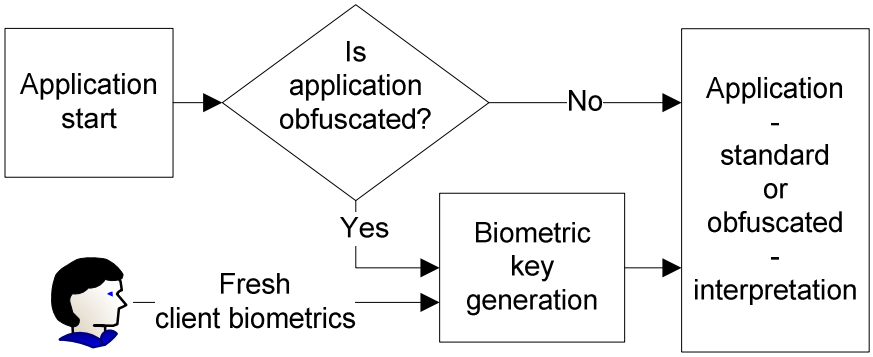


Figure 4. Application execution on client's mobile device

4. BOSSPRO IMPLEMENTATION AND RESULTS

4.1 Prototype implementation

To test this proposed biometric-secured obfuscated interpretation scheme, and to verify the practicality of this authentication, a prototype implementation, based on Android version 2.2 (Froyo), was developed.

Android is a software stack initially invented by Android Inc. and since 2005 developed by Google Inc and the Open Handset Alliance [19]. The fact that the complete software stack, including the operating system, middleware and important applications are available to the development community as Open Source makes Android a perfect candidate for this prototype development. Furthermore, Android is the fastest growing operating system used in mobile devices with enhanced sensors and capabilities. Note that this implementation does require the use of Smartphones so to capture and process the biometric data. It is expected that Android will surplus all other mobile operating systems by 2014 [20].

The Android operating system is based on a modified Linux kernel. System and client applications are executed/interpreted by the Dalvik virtual machine (DVM) which is part of the Android runtime and located in the Android system architecture above the Linux kernel. The DVM is a register-based virtual machine which interprets Dalvik byte code instructions generated by the "dx" tool from compiled Java language sources, i.e. each application, when started on an Android device, is first loaded from an .apk-file which contains the generated Dalvik byte code instructions. This is then verified and optimised before being interpreted by the DVM (Figure 5).

For our testing, to execute the biometric-secured obfuscated interpretation inside the DVM, the source code of the DVM was adapted; a new operating system image was compiled; and used with the Android emulator for testing this prototype implementation.

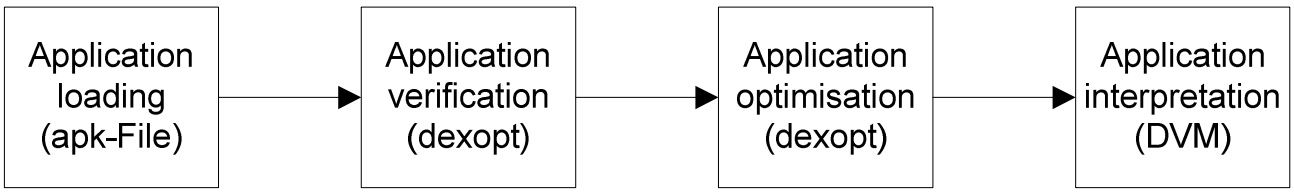


Figure 5. DVM steps during application execution

4.2 Byte code instructions for obfuscation

In order for an application to pass the compilation, loading, verification and optimisation steps of the Android DVM, not all byte code instructions can be substituted during the obfuscation process. For example, it is not possible to replace the "return-void" instruction because this is the only DVM instruction without parameters. Similarly, "iget*", "iput*", "invoke-*" or "invoke-*/range" instructions (Table 1) cannot be obfuscated because these instructions are automatically replaced by the DVM optimiser with other instructions, and thus are not available to the interpreter for de-obfuscation. Finally, instructions can be only replaced with other instructions, if and only if they have the same general return type as well as the same number and type of parameters. All other substitutions would not pass the verification stage of the DVM. For example the "add-int" instruction can only be substituted with 16 byte code instructions, the "if-eqz" with 6 instructions (Table 2).

Obviously, the security of obfuscated interpretation increases by increasing number of instructions that can be used for substitution of byte code instructions. Table 3 shows the total number of instructions for six Android applications. The first four applications (Browser, Contacts, E-Mail and Phone) are standard system applications available on all Android mobile devices, while the remaining two applications (PayPal and FXCM Mobile TSII (MarketSimplified Inc)) are "top-free in Finance" applications from the Android market. The total number of byte code instructions in the .apk-file of the application varies between 23.000 (Browser) and 100.000 (E-Mail) with around 60% of instructions that can be theoretically obfuscated.

Table 1. Instructions replaced by DVM optimiser

Instruction group	Instruction Mnemonic
iget*	iget, iget-wide, iget-object, iget-boolean, iget-short, iget-byte, iget-char, iget-short
iput*	iput, iput-wide, iput-object, iput-boolean, iput-short, iput-byte, iput-char, iput-short
invoke-*	invoke-virtual, invoke-super, invoke-direct, invoke-static
invoke-*/range	invoke-virtual/range, invoke-super/range, invoke-direct/range, invoke-static/range

Table 2. Possible instructions substitutions

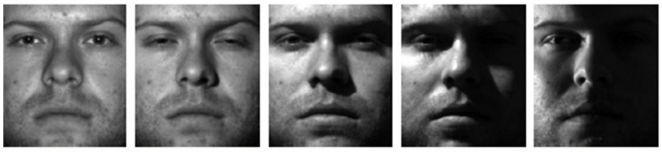
Instruction	Possible substitutions
add-int	add-int, sub-int, mul-int, div-int, rem-int, and-int, or-int, xor-int, shl-int, shr-int, ushr-int, add-float, sub-float, mul-float, div-float, rem-float
if-eqz	if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez

Table 3. Instructions available for obfuscation

Application name	Total # instructions	# instructions to obfuscate	% instructions to obfuscate
Browser	23000	14400	63%
Contacts	33800	22100	65%
E-Mail	99600	67700	68%
Phone	42200	25100	59%
Paypal	60000	38600	64%
FXCM Mobile	34300	20900	61%

4.3 Biometric generated keys

The BossPro prototype implementation of biometric key binding (generation) is based on face biometric as illustrated in Figure 2. The Extended Yale B database [21], which has 38 subjects and each one in frontal pose has 64 images captured under different illumination conditions, is used for the experiments. The images in the database are divided into five subsets according to the direction of the light-source from the camera axis. Figure 6 illustrates this variation for images of the same person in the database.



(a) Subset 1 (b) Subset 2 (c) Subset 3 (d) Subset 4 (e) Subset 5

Figure 6. Images for the same person in different illumination subsets

In the experiments, the first three images per client from subset 1 (the Yale-B database) were selected as the gallery set and all the remaining images were used for matching. Discrete Wavelet Transforms (DWTs) are selected as a facial feature extraction technique to be used efficiently on Smartphones. By selecting wavelet feature at the third level of decomposition, each face is represented by 504 value feature vector X, which is then converted to a binary string as described in [22]. By analysing the error patterns of inter- and intra-class variation of face images, it was concluded that 38% of the binary face feature vectors need to be corrected, i.e. 191 bits out of 504 bits. In other words, if the hamming distance between two binary feature vectors is less than 191, then the two feature vectors belong to the same individual. Otherwise, the two feature vectors are considered to be from two different individuals. To cope with intra-class variations of face samples, Reed-Solomon (RS) error correcting code algorithm is selected (version RS(511,129,191)), which takes a cryptographic key of size 129 bits as an input to produce a biometric lock of size 511 bits, and corrects up to 191 errors.

The experiments showed that the Equal Error Rates (EERs %) of biometric key binding is 0%, 0.9%, 1.33%, 15.48%, 17.15% for subset1, subset2, subset3, subset4, and subset5 of the extended Yale face dataset respectively based on Reed-Solomon ECC to retrieve a key of size 129 bits. It is worth mentioning that the 129 bit biometric key can be used as a seed to generate longer keys of any length based on techniques such as Linear Feedback Shift Registers (LFSR).

4.4 BossPro application development and byte code obfuscation

Android applications are written in Java, compiled with Java language compilers and then converted to Dalvik byte code by the Android “dx” tool. In BossPro prototype, the concept of Java Annotations is used to define which methods and/or complete classes should be obfuscated. Introducing the concept of “partial obfuscation” in the BossPro allows protection of nothing but the important parts and algorithms of an application. This will speed-up the DVM interpretation in applications which do not require complete code protection as the number of necessary de-obfuscation steps decreases. To obfuscate the byte code instructions, the Dalvik source code is de-compiled in the prototype by a disassembler. Figure 7 shows an example of a short java method with an “Obfuscate” annotation, two integer parameters and the resultant de-compiled Dalvik byte code.

As the number of instructions available for obfuscation varies from application to application, an instruction substitution key with a fixed length, as produced by a standard biometric key generator, is not applicable. A pseudorandom number generator (PRNG) with the biometric generated key as seed is used to produce a pseudorandom bit stream of the required length.

```
@Obfuscate()
private int doIt(int a, int b) {
  int c = a + b;
  if (c > 0) return c; else return a;
}
```

```
.method private doIt(II)I
  .annotation Luk/ac/buckingham/Obfuscate;
  add-int v0, p1, p2
  if-gez v0, :cond_1
  move v0, p1
  :cond_1
  return v0
.end method
```

Figure 7. Java source code (left) and Dalvik byte code instructions (right)

Depending on the byte code instruction and the next random bits from the PRNG output, an instruction from the same instruction group is selected. Figure 8 shows the obfuscation process for the two instructions “add-int” and “if-gez” (see Figure 7 and Table 2. First, the biometric key of the enrolled client is then extracted from the biometric key database and used as a seed to the PRNG. “add-int” is in step 2 the first instruction which needs to be obfuscated. The group size of possible substitutions for this instruction is 16 elements, i.e. requires 4 bits from the key stream. The first 4 bits from the stream are “0110” (or position 6 in the group, starting with 0). These bits are then used to index the substitution instruction “or-int”. The group of the second instruction “if-gez” contains 6 elements and therefore requires only the next 3 bits (“010”) from the key stream; resulting in the substitution instruction “if-ltz”. After all instructions are successfully obfuscated, the byte code is assembled again. This "obfuscation and protected Android application file" can be then distributed to the client.

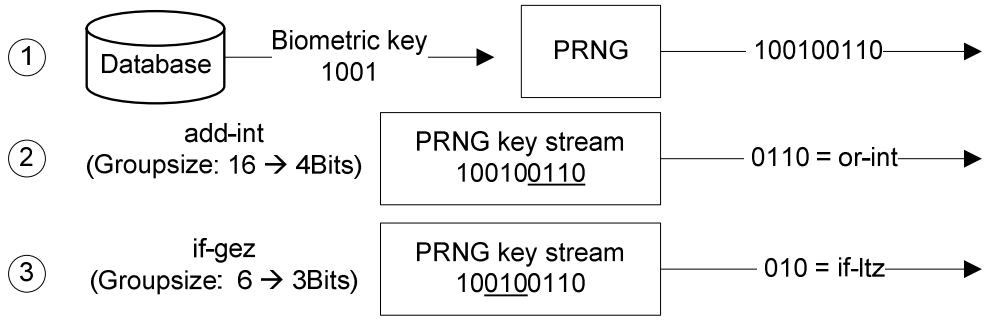


Figure 8. Instruction obfuscation process

4.5 Application execution and de-obfuscation process on the mobile device

Upon receiving the obfuscated application from the authenticator, the client installs it on his mobile device. Once the client starts this protected application, the DVM on the mobile device triggers a process to capture fresh biometric information of the client. A biometric key and its pseudorandom bit stream are then calculated/generated in similar steps to the key generation process at the authenticator side (cp. section 4.4). The resultant PRNG output is then used to de-obfuscate the application code during interpretation. Based on the next PRNG input value and the following obfuscated instruction read from the protected byte code, the original instruction is determined/obtained by reversing the used obfuscation rule (Table 2). The resultant de-obfuscated code is then ready to be executed by the DVM.

5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

In conclusion, this paper proposes to combine biometric-based key generation with obfuscated interpretation to prevent the illegitimate execution of applications as well as to protect the software against reverse-engineering. This is

particularly aimed at, but not limited to, application software installed on mobile devices with enhanced capabilities and sensors such as Smartphones or Tablet-PCs. BossPro can be utilised in a similar way to all kinds of software programs running on standard PCs. Obfuscating the program instructions with a client's specific biometric key shall tightly binds the genuine client to the application and hence eliminates the possibility that an attacker is able to use this protected application. This becomes more and more important as the use of mobile devices to perform financial/business or other security related transactions grows.

The implemented prototype of BossPro based on Android Froyo shows clearly the practicality of this proposed scheme. That is the advantages and benefits of this tight combination of biometric authentication and software protection through obfuscated interpretation. The Dalvik virtual machine (which runs all system and client applications on Android based mobile devices) was modified to test the obfuscated interpretation in a real operating system environment. An Android system image with the adapted DVM was generated and used in the emulator based trials and experiments. Analyses of many Android build-in and market-place applications as well as the Dalvik byte code structure shows that around 60% of all byte code instructions can be statistically obfuscated. Also note that instructions can be replaced only with similar ones from the same instruction group. Otherwise the byte code verifier or optimiser would recognise the incorrect program and an application installation on the mobile device would not be possible. Also, the group size of similar instructions varies for the Android Dalvik byte code between two and sixteen elements. This results in an immense number of possible substitution combinations. For example, a simple method with only 20 instructions (10 instructions from the “add-int” and 10 from the “if-eq” group) would already result in 6.6*1019 possible combinations. The fact that a standard Android application has several thousand byte code instructions makes it very difficult to understand the real semantics of a program without having the correct de-obfuscation key. Although it would be possible for an attacker to start a program protected by BossPro, as the obfuscated program is still a valid application, the attacker cannot be sure about the program behaviour or results, or if the program terminates correctly. However, it is more likely that the program will crash and produce no meaningful output at all.

Since BossPro adds with the required byte code deobfuscation process an additional step to the application interpretation and therefore increases the application execution time. As the methods and classes and consequently the resultant number of byte code instructions to be deobfuscated can be precisely specified during the application development process, the run-time overhead can be adjusted to the desired security level of the application. Trails on the emulator showed that the introduced overhead by BossPro is not noticeable for any client in an authentication application scenario. However, full time and CPU overhead measurements are planed when the full actual implementation has been completed and tested on Android mobile devices.

5.2 Future Work

Work on the biometric-secured obfuscation interpretation is ongoing to further analyse and enhance the security of BossPro. As a first step, the authors will extent the prototype and implement all possible instruction substitutions in the obfuscation process as well as inside the Dalvik virtual machine. Furthermore, real-world experiments and trials on Android mobile devices will be carried out, which requires installation of the adapted Android operating system on real hardware. The authors will also investigate, how the integration of present location and real-time into the key generation and obfuscation process can further eliminate various possibilities of application misuse, i.e. by employing the current location of the mobile device determined by the GPS receiver into the obfuscation algorithms. This shall eliminate various types of "distance attacks" which are a main threat to mobile devices and mobile based applications.

To further verify and increase the security of BossPro, the results of various de-compilation and reverse-engineering tools for Java and Dalvik byte code, e.g. “undx” or “Dex2Jar” on the biometric-secured obfuscation applications will be analysed. However, first reverse-engineering experiments clearly showed that these programs are not able to restore the original semantics of the obfuscated applications. Finally, the combination of biometric-secured obfuscated interpretation with other software protection techniques, e.g. control flow obfuscation [23] or opaque constructs [24], will be investigated. It is expected, that these techniques can be easily used together and that a combination will not negatively affect the obfuscated interpretation. In contrast, they should further enhance the security of BossPro as they build a first line of defence against attacks and even increase in some cases the number of instructions which then can be obfuscated.

REFERENCES

[1] B. S. Alliance, "Seventh annual bsa/idc global software 09 piracy study," (2010)

[2] S. Z. Li and A. K. Jain, [Encyclopedia of Biometrics], Springer US, (2009)

[3] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," IEEE Transactions on Software Engineering, 28, 735–746 (2002)

[4] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," Electronic Commerce Research, 6(2), 155–171 (2006)

[5] W. Zhu and C. Thomborson, "Algorithms to watermark software through register allocation," Digital Rights Management. Technologies, Issues, Challenges and Systems ser. Lecture Notes in Computer Science, 3919, 180–191 (2006)

[6] G. Myles and H. Jin, "Self-validating branch-based software watermarking," Information Hiding ser. Lecture Notes in Computer Science, 3727, 342–356 (2005.

[7] D. Aucsmith, "Tamper resistant software: An implementation," Information Hiding ser. Lecture Notes in Computer Science, 1174, 317–333 (1996)

[8] P. Wang, S. Kang, and K. Kim, "Tamper resistant software through dynamic integrity checking," Proc of the SCIS, (2005)

[9] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, (1997)

[10] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in Lecture Notes in Computer Science, 1–18 (2001)

[11] A. W. Appel, "Deobfuscation is in np," (2002)

[12] A. Monden, A. Monsifrot, and C. Thomborson, "A framework for obfuscated interpretation," ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, 7–16 (2004)

[13] X. Zhang, F. He, and W. Zuo, "A framework for mobile phone java software protection," ICCIT '08: Proceedings of the 2008 Third International Conference on Convergence and Hybrid Information Technology, 527–532 (2008)

[14] Y. Zeng, F. Liu, X. Luo, and C. Yang, "Robust software watermarking scheme based on obfuscated interpretation," Proc 2010 International Conference on Multimedia Information Networking and Security, 671–675 (2010)

[15] F. Hao, R. Anderson, and J. Daugman, "Combining cryptography with biometrics effectively," IEEE Transactions on Computers, 1081–1088 (2006)

[16] K. Nandakumar, A. Jain, and S. Pankanti, "Fingerprint-based fuzzy vault: Implementation and performance," IEEE Transactions on Information Forensics and Security, 2(4), 744–757 (2007)

[17] H. Al-Assam, H. Sellahewa, and S. Jassim, "A lightweight approach for biometric template protection," Proc SPIE 7351, 73510P.1–73510P.12 (2009)

[18] A. Teoh, Y. Kuan, and S. Lee, "Cancellable biometrics and annotations on biohash," Pattern recognition, 41(6), 2034–2044 (2008)

[19] O. H. Alliance, "Open handset alliance." http://-www.openhandsetalliance.com/

[20] Gartner Inc., "Forecast: Mobile communications devices by open operating system, worldwide, 2007-2014 (2010)

[21] A. Georghiades, P. Belhumeur, and D. Kriegman, "From few to many: Generative models for recognition under variable pose and illumination," IEEE Transactions on Pattern Analysis and Machine Intelligence, 23, 643–660 (2001)

[22] A. Jin, D. Ling, and A. Goh, "Biohashing: two factor authentication featuring fingerprint data and tokenised random number," Pattern Recognition, 37(11), 2245–2255 (2004)

[23] T. Hou, H. Chen, and M. Tsai, "Three control flow obfuscation methods for java software," IEE Proceedings-Software, 153(2), 80-86 (2006)

[24] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," Conference Record Of The ACM Symposium On Principles of Programming Languages 1998, 184–196 (1998)