Using Virtual Machine Protections to Enhance Whitebox Cryptography

Joseph Gan, Roddy Kok, Pankaj Kohli, Dr. Yun Ding, Benjamin Mah
V-Key Pte. Ltd.
Singapore, Singapore
{joseph.gan, roddy.kok, pankaj.kohli, yun.ding, benjamin.mah}@v-key.com

***Abstract*—Since attackers can gain full control of the mobile execution environment, they are able to examine the inputs, outputs, and, with the help of a disassembler/debugger the result of every intermediate computation a cryptographic algorithm carries out. Essentially, attackers have total visibility into the cryptographic operation.**

Whitebox cryptography aims at protecting keys from disclosed in software implementation. With theoretically unbounded resources a determined attacker is able to recover any confidential keys and data. A strong whitebox cipher implementation as the cornerstone of security is essential for the overall security in mobile environments.

Our goal is to provide an increased degree of protection given the constraints of a software- solution and the resource-constrained, hostile-host environments. We seek neither perfect protection nor long-term guarantees, but rather a practical level of protection to balance cost, security and usability. Regular software updates can be applied such that the protection will need to withstand a limited period of time. V-OS operates as a virtual machine (VM) within the native mobile operating system to provide a secure software environment within which to perform critical processes and computations for a mobile app.

***Keywords*—Virtual Machine Protections (VMP), Code Obfuscation, Data Obfuscation, Anti-Reverse Engineering, Anti-Debugging, Whitebox Cryptography (WBC), Software Renewability, Mobile Code, Software Tamper Resistance, Fingerprinting, Software Licensing.**

I. INTRODUCTION

In open literature, the strongest whitebox cipher implementations [3], [5] do not provide protections that are adequate for memory and performance constrained mobile environments. Moreover, given that attackers are increasingly employing dynamic attacks against software [6], the research papers do not address how to protect the concrete whitebox implementations in a hostile environment.

In this paper, we present a strategy that uses Virtual Machine Protections as defence-in-depth to strengthen whitebox cryptography implementations. A virtual machine, V-OS, is designed with multiple layers of protection that adopts a practical whitebox AES implementation, and enhances it with multiple V-OS VM protection techniques. The use of our own virtual machine allows for control over the operating environment in a way that allows the implementation of a number of protection mechanisms in the virtual operating environment that would not otherwise be possible in iOS or Android using a user-space application. The virtual machine integrates the whitebox AES implementation with the virtual machine protections so that a replacement attack cannot be carried out [7]. We have built and provided an initial version of this in a commercial product, and are continuing to implement additional protections and strengthen the existing mechanisms. V-OS provides cryptographic functions within the virtual machine that can be called either by native Android and iOS mobile applications, or by virtual V-OS Trusted Applications running within the virtual machine. V-OS can also support software updates for the AES whitebox implementation as recommended by M. Jakobsson and M.K. Reiter [8].

In Section II we describe the limitations of the current table-based implementation of AES proposed by S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot in 2002 [1]. We also describe other whitebox implementations that we consider unsuitable for mobile usage at this time because of their comparably higher memory demand.

In Section III, we introduce a hardened virtual machine architecture, V-OS, which we use as the foundation for protecting a whitebox implementation. V-OS is designed to operate as a virtual Secure Element that complies with Global Platform standards and architectures for financial transactions [11]. The objective is to provide a tamper-resistant software-based root of trust that can then enable a variety of mobile use-cases for identity, authentication, and authorisation.

In Section IV we elaborate on how the data in the whitebox AES algorithm (e.g., look-up tables and S-box) is obfuscated to make the data indistinguishable from random permutations. Moreover, they are dynamically addressed to hide the algorithm's execution flow. We describe how this compels an attacker to reverse engineer the entire code base or significantly larger code segments in an attempt to achieve their goals.

In Section V, we describe how the firmware byte-code is obfuscated at multiple levels, source code, and assembly code. To protect the firmware binary from static analysis, part of the source code is itself dynamically modified at byte-code level during run-time prior to use. Prior to assembly, dead code and random data is injected to the assembly code. The order of data and code in the binaries are also randomised prior to

978-1-4673-7094-3/15 $31.00 © 2015 IEEE
DOI 10.1109/SPRO.2015.12

assembly. Code obfuscation is designed to obfuscate across the entire firmware, so that an attacker needs to reverse engineer the entire binary with each new firmware.

In Section VI, to prevent attackers from attacking V-OS on their own devices, we describe further protections, including V-OS anti-debugging, memory encryption and native code obfuscation, which make it harder for attackers to achieve their goals of reverse-engineering the whitebox system. We also describe how V-OS is designed to be non-portable; it should only run on the primary device it is installed on, much like a physical entity. The V-OS firmware can be bound to a device using a device fingerprint as an additional precondition for the obfuscation and deobfuscation of code and data.

We evaluate the performance and security characteristics of V-OS in Section VII, and propose ways in which this cryptosystem can be further improved in Section VIII, including strengthening the ties between the WBC and VMP, leveraging additional protections in the VMP, exploring the possible use of custom S-boxes, and studying how these protections can be applied to public key cryptography.

II. Limitations of Current Whitebox AES Implementations

This section will briefly cover the original whitebox AES implementation by S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot in 2002 [1], as well as consider several other alternatives.

A. Whitebox Scenario

In a typical blackbox attack scenario, an attacker attempts to recover the key used in an implementation of a cryptosystem by looking at a long paired list of plaintext and ciphertext, and deducing or guessing the key used. In the whitebox scenario, an attacker has access to the execution environment including the content of the runtime memory, firmware binary, and may have made inroads in deducing the source code. In the worst case, the attack has the full source code for the implementation, for which recovering the key used in a standard AES implementation is a trivial exercise.

Whitebox AES is a mathematical approach to implementing AES with a given key that makes it difficult to recover the key even if the source code is known. This can be loosely classified as a form of code obfuscation, but with more formal mathematical foundations, so as to obtain a complexity measure of the level of difficulty.

B. Original Table-Based AES Implementation

The above definition of AES is implemented as a series of look-up tables, with every table used exactly once. To transform the AES implementation without modifying the output, randomly chosen self-cancelling input and output encodings and mixing bijections are inserted at the start and end of consecutive tables. From the randomised tables, it will be difficult to deduce the inserted encodings directly, which are necessary for deducing the embedded key. Note that this design ensures that the blackbox behaviour of an AES implementation remains unchanged even after transformation. This means that a standard blackbox AES implementation can be used to decrypt ciphertexts resulting from a whitebox AES implementation.

An attacker who has successfully recovered all the tables used in the implementation will still be faced with a mathematical problem of guessing the encodings and bijections used in order to retrieve the keys.

TABLE I. Original Whitebox AES Table Sizes

Mapping	Original Size	Original No. of Tables	Original Total Size
8-bit to 32-bit	1024 bytes	288	288 KB
8-bit to 4-bit	128 bytes	1728	216 KB
8-bit to 8-bit	256 bytes	16	4 KB
TOTAL SIZE OF TABLES			508 KB

Each such implementation will require around 500 KB of table data (see TABLE I) per transformed AES implementation, while the time complexity of the implementation is approximately 2^{20}.

With access to the source code of such an implementation, using the BGE attack by O. Billet, H. Gilbert, and C. Ech-Chatbi from 2004 [2], it is known that a work factor of less than 2^{30} is sufficient to recover the keys used in such a whitebox AES implementation.

C. Alternative Whitebox Implementations

In 2009, Y. Xiao and X. Lai [3] proposed a new whitebox AES implementation designed to be resistant to the BGE attack. This approach required 20502 KB, and had an increased time complexity of 2^{24}. Subsequent cryptanalysis by Y. De Mulder, P. Roelse, and B. Preneel in 2012 [4] showed that a work factor of approximately 2^{32} would be sufficient to recover the AES key.

S. Yang, Q. Liu and Q. Zhao proposed another approach in 2013 [5] based on SHARK that required less table data of 14339 KB and had a higher security level, but was much slower. However, the amount of table data required is still largely infeasible for a constrained mobile or embedded environment.

III. Strengthening Whitebox Resistance Using VMP

Whitebox implementations for mobile environments need to be small enough to be feasible, but current whitebox implementations are either too large to be practical for constrained mobile environments or too weak by themselves.

To strengthen the implementation of a weaker whitebox implementation that can be small enough to be used in a mobile environment, we have developed a hardened virtual machine, V-OS, that adds considerable protection for both the data (look-up tables) and the code (addressing instructions) to make the recovery of the source code itself a difficult task. The use of a hardened virtual machine provides increased isolation for the operating environment where the whitebox cryptographic implementation is executed, allowing for the implementation of additional protection methods around the

whitebox solution. Some of the data and code obfuscation methods that can be used are described in Sections IV and V.

The overall architecture of V-OS is described in Fig. 1. Section VI describes a number of protection mechanisms that help to isolate V-OS from both the mobile application as well as the native runtime environment (RTE), which is currently on iOS and Android. These are designed to provide protection against both reverse engineering and dynamic attacks on the cryptosystem. V-OS then provides specific interfaces to interact with the underlying runtime environment or for the mobile application to interact with V-OS.

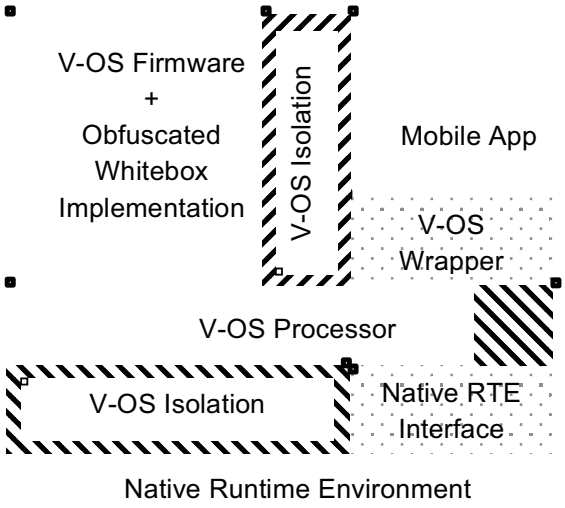

Fig. 1. Overall Architecture of V-OS

This isolated virtual machine operating environment, similar to a hardware secure element, also allows the use of a whitebox solution in higher-security environments where certifications such as Common Criteria and FIPS 140-2 Levels 2 and above require the cryptographic implementation to be executed in a hardened isolated operating environment.

IV. Data Obfuscation

A. Whitebox AES Data

The V-OS firmware binary can contain both the code and data to be obfuscated, and can be updated in a fully obfuscated form. This allows for the lookup table randomisation and AES key to be replaced, depending on validity periods defined in the customer's security policy around V-OS use. As code and data obfuscation is performed across the entire firmware file, there will be significant differences in the firmware binaries, making it difficult to glean useful information on the location of the algorithm by comparing updates.

As the lookup tables for the whitebox AES implementation are normally chosen to ensure that every table is used at most once, there is considerable redundancy in the form of replicated tables prior to the insertion of random bijections. Besides that, the different entropy levels and structures of the three table types make it easier for an attacker to distinguish between the tables, and accelerate the reverse engineering process.

In the event the code is indistinguishable to the attacker, the attacker can attempt to trace the execution path of the whitebox AES algorithm in the static code and painstakingly map the data to the rounds to which each lookup table belongs to. While a few thousand lookups are needed per execution, this is by no means an insurmountable task if the code flow is not obfuscated.

To deny the attacker a straightforward and certain trace of the execution path, the data are stored obfuscated in static form, and are deobfuscated by a process triggered outside the execution of the whitebox AES algorithm. This ensures that tracing the execution of the algorithm itself is insufficient to recover the source; debugging and tracing the execution of all processes will be necessary to mount an attack.

B. Data Masquerade

To make the lookup tables indistinguishable from each other, each of the larger tables of 1024 bytes are broken up and stored as four 256-byte arrays. Two smaller 128-byte tables are combined to form a 256-byte array. The existing 256-byte tables exist as 256-byte arrays. As the tables have differing entropy patterns, we can use a data obfuscation routine to modify the byte values to increase the entropy for all the tables.

With this modification to the data set, the lookup tables appear as 2032 256-byte arrays in the firmware binary. To take the indistinguishability further, the addressing to each table is dynamic in that it may be modified at runtime. The code size taken per table is 4 bytes for the address and 8 bytes for the V-OS instruction, so the additional code size is 2032 * 12 bytes = 24 KB. The final configuration of obfuscated tables is as follows (see TABLE II):

TABLE II. Obfuscated Whitebox AES Table Sizes

Mapping	Obfuscated Size	Obfuscated No. of Tables	Obfuscated Total Size
8-bit to 32-bit	256 bytes	1152	288 KB
8-bit to 4-bit	256 bytes	864	216 KB
8-bit to 8-bit	256 bytes	16	4 KB
TOTAL SIZE OF TABLES		2032	508 KB
Size used for addressing tables in firmware			24 KB
TOTAL SIZE INCLUDING ADDRESSING			532 KB

C. Table Address Obfuscation

Having all lookup tables exist as byte permutations, addresses of the tables may be permuted prior to restoring the data values. Furthermore, without prior knowledge, it is difficult to determine if the current addressing of tables is the correct state.

The deobfuscation process not only restores the data values to its original form for the correct execution of the algorithm, but also restores the addresses of the tables. The deobfuscation process can be implemented as multiple independent processes, so that no fixed call sequence is necessary. To determine if the whitebox AES is in the correct

state for execution, it suffices to ensure that every deobfuscation sub-process has been executed.

V. Code Obfuscation

A. V-OS Firmware Byte-Code

V-OS is implemented in C, which is then compiled into assembly code before being assembled into byte code. V-OS currently uses a Reduced Instruction Set Computing (RISC) set of instructions that is interpreted within a virtual processor, although there is little performance implication if this is changed to a Complex Instruction Set Computing (CISC) architecture or if more instructions are added for improved functionality or performance. At the byte code level, data and code are both addressed using the byte representation of their addresses, and reverse engineering of the source code is done via following the addressing of the functions and data used in each event. The techniques described in this section have been selected to increase the difficulty of an attacker in analysing the compiled V-OS code, especially in the analysis of the table data lookups, without significantly slowing the performance of the cryptographic functions, although there are many other well-known source code and binary obfuscation techniques that also can be applied within the V-OS firmware codes.

B. V-OS Function Call Obfuscation

Static analysis involves studying the firmware code, typically from a known point of entry (e.g. boot), to trace the byte code that is called. The aim of generating obfuscated function calls is to modify the data and function addressing at runtime, so that tracing an expected execution path on the static code alone is insufficient, as the execution path is changed not at the source code level but at the byte code level. For example (see Fig. 2):

```
function1();  // address = 0x00274100

function2();  // address = 0x00274200

modify_addr() {
        // modify the address for function1
        function1 += 0x0100;
}

void main() {
        // modifies the address of function1
        modify_addr();

        // calls function2 instead of function1
        function1();
}
```

Fig. 2. Example of V-OS Function Call Obfuscation

Having multiple dynamic function call generation pieces implies that all functions in an execution path may impact the algorithm used. An attacker will thus have to trace all execution paths or reverse engineer the entire binary to discover exactly the state of the algorithm at the time of execution.

C. V-OS Assembly Code Obfuscation

After the source code has been compiled into assembly codes, more obfuscation can take place. Obfuscation at this level has a more direct effect on the binary that the attacker sees.

An additional obfuscation that can be performed at the assembly level is the insert of dead code, which is code that has no effect during execution or is never executed, e.g. *mov r1, r1*.

In a similar fashion, dummy code or data are injected at this point, to misdirect attackers. Compared to dead code, dummy code or data usually refers to bytes that resemble actual byte codes, but are also not actually used.

To guard against pattern matching, commonly used during reverse engineering to save effort on duplicated code, certain C code will generate different polymorphic byte code equivalents. For example (see Fig. 3):

```
mov r1, 0 → mov r2, 0

mov r1, 0 → xor r1, r1
```

Fig. 3. Examples of V-OS Assembly Code Obfuscation

VI. Further Protections

In addition to the obfuscation of code and data, V-OS is deployed with additional protections that limit the scope for attempts to reverse engineer the code.

A. V-OS Device Fingerprint

In V-OS, device fingerprint (DFP) is used to bind V-OS to a specific device, and prevent V-OS from running and being reverse-engineered on a device chosen by the attacker. The objective of the DFP function is to prevent an attacker from cloning the V-OS firmware and spoofing the DFP attributes on another device.

The DFP is defined as a high entropy variable that takes the aggregate value across several unique device identifiers and an application identifier. The inclusion of the application identifier is to prevent an attacker from using a V-OS firmware designed for one application in another, even on the same device. The device fingerprint is computed as a hash of a concatenation of the single application identifier, followed by the device identifiers in order of decreasing entropy. These device identifiers include globally unique identifiers such as the CPU serial number and IMEI in Android, and the motherboard serial number and processor identifiers in iOS.

To trust the device fingerprint, various techniques have been implemented to check the integrity of these identifiers, by verifying that the sources of these device identifiers themselves are trusted. These include verification of function pointers, cross-checking and validation of critical functions against known libraries and frameworks, detection of known native hooking techniques in both iOS and Android native libraries, and detection of Android Dalvik runtime manipulation via Java method validation.

When the V-OS virtual machine starts up, the DFP is collected and verified from within the V-OS firmware. Combined with the unique identifier of a personalised V-OS, the DFP is used to derive a secret device key, which can also be used to generate One Time Passwords for device authentication.

For personalised V-OS, the DFP can be used to generate an additional input to the obfuscation of whitebox AES data and code, so that its absence will deny an attacker from retrieving the algorithm source.

B. V-OS Anti-Debugging

V-OS implements debugger detection at multiple layers to provide a comprehensive detection mechanism in order to detect and block debuggers. In the primary layer, it detects if any debugging daemons such as Android's *adbd* or iOS/Android's *gdb* and other *ptrace*-based debuggers are running and actively debugging the application, using both signature- and behavioural-detection. This is performed when an applications starts and is additionally verified whenever the V-OS firmware is called. While the application is running, V-OS employs additional threads to detect debuggers in the background at regular intervals. This is done to detect debuggers such as *gdb* that may attach to the application while it is running. It observes the processes and sub-processes to detect if a debugger is being attached.

The real time debugger detection uses background events to detect and defend against debuggers such as DDMS-initiated attacks. V-OS verifies the Dalvik Virtual Machine (DVM) to detect debuggers that try to attach to the Dalvik runtime to debug or dump the process. It monitors Android kernel for any changes to detect memory dump attempts that may be initiated from the kernel. The debugger detection is performed in a way such that it does not affect the performance of the application or the battery life of the device. We have also determined that there are similar techniques available on iOS operating systems.

Anti-emulation is a technique we have experimented with to provide additional protections against dynamic analysis, including software debugging but also extending to hardware debugging and malicious virtual machine emulation. The idea is to perform self-benchmarking of critical portions of code in order to ensure that debugging breakpoints have not been inserted, and an attacker has not tampered with critical portions of the virtual processor such as the native system calls, using timing analysis from within the V-OS firmware to detect such an attack in progress. However, while this concept works against some attacks, we have found this difficult to implement in production due to the native processor speed variation both between mobile devices as well as within an execution session due to multitasking.

C. V-OS Memory Encryption

To limit the exposure of sensitive data in memory, the data are encrypted when written to memory, and decrypted only just before use. The encryption here can be optimised for speed, and retards reverse engineering by making the process of memory dumping much more onerous.

We have evaluated several ways in which the memory encryption keys used by the V-OS virtual processor can be protected. The fastest is to store the encryption keys in the global processor memory, which we call global memory encryption. These keys can be generated at runtime and differentiated for each portion of memory. This requires that we assume the virtual processor to be well protected against dynamic memory attacks such as debugging and memory dumping. The advantage of this approach is that it can be applied to both virtual codes and data with less than 10% performance slowdown.

For more secure encryption of sensitive data such as other cryptographic keys and information when in use, stronger techniques can be used. One technique we have implemented is to hide a portion of the encryption key used within the upper bits of the virtual memory address of the data variable, which we call discretionary memory encryption. When the virtual firmware uses this virtual memory address to refer to the data, the V-OS processor automatically applies the encryption to the data. This approach protects the fact that specific data is encrypted from the attacker as there is no information in and around the data that indicates it is specifically encrypted, other than the entropy of the data which would original be high if it was an encryption key. This approach also continues to encrypt this data even if the global memory encryption is overcome. The attacker would need to go through the time-consuming process of reverse-engineering the entire firmware in order to determine which pieces of data are encrypted in this manner and what the encryption keys are. We can also use stronger encryption with this technique as it is only used for small pieces of sensitive data rather than for all code and data, and the additional performance impact is negligible.

D. Tamper Response

V-OS is also able to provide tamper response mechanisms that effectively prevent an attacker from retrieving the cryptographic keys. We have implemented mechanisms that detect tampering of V-OS, by detecting attacks such as code modification, debugging or function hooking. We have also developed mechanisms in V-OS that allow it to boot up securely, by performing self-tests and integrity checks of the virtual processor and firmware. This protects V-OS from tampering and also makes it difficult for an attacker to try to bypass the protection mechanisms scattered throughout the system.

When tampering is detected, we cause the firmware to follow a code path or set certain data to be different from the untampered system. This can be triggered silently so that an attacker will not realise that the tampering has been detected unless he performs a full reverse engineering and code analysis of the virtual processor and firmware. The objective of this technique is to trick the attacker into following fake code and data flows that either lead nowhere, or that use fake keys whose use can later be detected by a remote system outside of V-OS so as to disable the usage of the real keys.

E. Native Code Obfuscation

V-OS's operation is that of a binary file containing instructions that are executed by a virtual processor. The virtual processor, running in the native Android or iOS

ARM/x86 environment, has the task of interpreting the instruction byte code in the V-OS firmware binary and mapping it to the required operations in the native environment.

The reduced virtual processor instruction set used by V-OS contains approximately 32 unique instructions, which may take on arbitrary byte values. As the determination of this instruction set is the first step to reverse engineering the firmware binary, this instruction set will be randomised and uniquely determined for each customer. In this way, if one customer recovers the instruction set mapping, the security of another customer is not affected. To guard against frequency analysis on the instruction set, the number of distinct bytes used to represent each instruction is proportional to its frequency of use in an average code.

In addition, native code obfuscation is applied to the virtual processor code to resist reverse engineering. This helps to prevent an attacker from analysing the specific implementation of the virtual machine processor's random instruction set. We have evaluated and integrated the Obfuscator-LLVM compiler port [9], although our process allows for any source code or binary obfuscation tool to be used. As native code obfuscation leads to a significant slowdown of the V-OS processor, especially the stronger techniques such as control flow flattening, we have chosen to apply these to selected portions to deter reverse engineering while minimising the performance impact. These help to protect against execution and call graph analysis from reverse engineering and debugging tools.

VII. Evaluation

A. Security

The security of a whitebox implementation relies on the attacker being able to determine the table data used so that the cryptanalysis can proceed. By combining whitebox techniques with virtual machine protection, the attacker is forced along a manual path of reverse engineering or dynamically tracing the virtual firmware execution in order to try to bypass the various layers of protection to retrieve the table data. Given that the virtual machine allows protections to be implemented at multiple levels, this will be a very time-consuming process, and we consider the cryptosystem to be successful if an attacker is forced to attempt to perform such a full reverse engineering manually. Given the tamper detection and response mechanisms in V-OS, the attacker is likely to trigger an incorrect code path if a combined manual and automated process is used to try to bypass the protections, which would result in a failure to retrieve the correct table data.

If the attacker chooses not to try to trace the execution, he can try to identify the table data through the entropy of the data scattered throughout the firmware. Even assuming that the portions of the table data can be identified perfectly through statistical cryptanalysis, there are now 2032 256-byte arrays whose order or allocation is undetermined. This adds an additional work factor of 2032! which is unfeasible to attack directly. Even assuming further weakening by an attacker who is able to distinguish between various types of table data from the different mapping tables, there are still far too many permutations for an attacker to try to cryptanalyse without a full analysis of the virtual firmware codes.

B. Performance

The current virtual machine implementation adds a time complexity of less than 2^7 over an unprotected native implementation and very little additional data above and beyond the underlying whitebox cryptographic table data used. This includes protections embedded within the virtual processor and firmware such as memory encryption and the code obfuscations we have selected. We consider this performance to be acceptable for securing the root keys in a mobile use case, although we expect that the current performance can be optimised further, and this can also be tuned by adjusting both the V-OS and native code obfuscation performed.

The additional protections such as anti-debugging and device fingerprinting do not add perceptible slowdown to the cryptosystem.

The memory overhead of using V-OS can be as small as 1MB when only AES is used, since both the virtual processor and firmware are written in embedded C. This includes the 508 KB for the whitebox AES table data in the virtual code. V-OS can allocate more memory when additional V-OS cryptographic functionality is required or when multiple virtual V-OS Trusted Applications are desired for the use case.

In terms of the code sizes, the V-OS virtual processor currently uses about 100 KB, while the V-OS firmware including the whitebox AES implementation is approximately 50 KB excluding the obfuscated table data embedded within the firmware binary.

VIII. Future Work

A. Strengthen Ties Between WBC and VMP

The current implementation uses the virtual machine codes and data to help prevent reverse engineering, and help cryptanalysis, of the whitebox table data. We propose to explore how these protections could be leveraged even more tightly with the table data to further prevent cryptanalysis, as well as to continue to quantify the work factor that an attacker would face in attacking such an enhanced implementation.

B. Leveraging Additional Protections in VMP

While the current implementation already implements a number of protection mechanisms to harden the virtual machine environment, we propose to explore how additional protections such as anti-debugging, anti-emulation, and memory encryption could be strengthened to make the protections even harder to bypass.

We propose that further work in anti-emulation shall study how the timing analysis techniques can be applied on diverse mobile devices with irregular processing speeds, in order to make this technique more stable in production.

The discretionary memory encryption can be extended to custom discretionary memory encryption techniques, whereby the encryption key to be used is also stored within a V-OS

firmware function that is called whenever the processor is directed to encrypt a portion of memory. For enhanced security, the memory encryption function can be implemented as a V-OS firmware function so that both the memory encryption key and implementation can be protected within the V-OS firmware.

C. Custom S-Box

As an additional option, the S-Box in the V-OS whitebox AES implementation can be replaced with another custom keyed S-Box that meets the requirement for linear and differential cryptanalytic resistance [10]. The BGE attack relies on approximating an affine transformation that characterises the randomised encodings and bijections, and validating the guess. Without knowledge of the underlying S-Box, the attacker needs to guess the S-Box used before attempting to recover the embedded key. In the case of a wrong S-Box guess, the attacker will not be able to recover the correct AES key. This in effect increases the work factor of the attack by the diversity of S-Boxes used.

D. Use of VMP for Public Key Cryptography

While this paper has focused primarily on how the protections of a virtual machine can strengthen a symmetric cryptographic system such as AES, many of the protections can apply to public key cryptography. We propose to study how a whitebox RSA or ECC implementation could benefit from the code and data obfuscation, as well as the virtual machine protections of the V-OS architecture.

IX. Conclusion

In this paper, we propose the use of a hardened virtual machine that allows for the implementation of a simpler and smaller whitebox cryptography implementation in order to fit the size-constrained mobile or embedded environment. This strengthens a whitebox cryptography implementation with virtual machine protections. Further work should be focused on strengthening the virtual machine protections and tying these protections more closely with the whitebox table data, as well as exploring the use of custom AES S-boxes in such an implementation.

Although this paper has focused on combining table-based AES whitebox cryptography, many of the protections described apply to other cryptographic protection requirements. These protections have been applied to other use cases such as public key cryptography, license control, and native application tamper protection by building cryptographic functionality on top of the V-OS virtual machine.

Acknowledgment

The authors are grateful to the anonymous reviewers for providing insightful comments and guidance for improving this paper.

References

[1] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, “White-Box Cryptography and an AES Implementation,” in 9th Annual Workshop on Selected Areas of Cryptography (SAC 2002), Lecture Notes in Computer Science 2595, Springer-Verlag, 2003, pp. 250-270.

[2] O. Billet, H. Gilbert, and C. Ech-Chatbi, “Cryptanalysis of a White Box AES Implementation,” in Selected Areas of Cryptography (SAC 2004), Lecture Notes in Computer Science 3357, Springer-Verlag, 2005, pp. 227-240.

[3] Y. Xiao and X. Lai, “A Secure Implementation of White-Box AES,” in 2nd International Conference on Computer Science and its Applications (CSA 2009), IEEE, 2009, pp. 1-6.

[4] Y. De Mulder , P. Roelse, and B. Preneel, “Cryptanalysis of the Xiao – Lai White-Box AES Implementation,” in 19th Annual Workshop on Selected Areas in Cryptography (SAC 2012), Lecture Notes in Computer Science 7707, Springer-Verlag, 2012, pp. 34-49.

[5] S. Yang, Q. Liu and Q. Zhao, “A Secure Implementation of a Symmetric Encryption Algorithm in White-Box Attack Contexts,” Journal of Applied Mathematics 2013, Hindawi, 2013, pp. 1-9.

[6] P. Biondi and D. Fabrice, “Silver needle in the skype,” Black Hat Europe (Amsterdam, the Netherlands, 2006).

[7] S. Ghosh, J. Hiser, and J. W. Davidson, “Replacement Attacks Against VM-protected Applications,” in Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, ACM, 2012, pp. 203-214.

[8] M. Jakobsson, M. K. Reiter, “Discouraging Software Piracy Using Software Aging,” in Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001, Lecture Notes in Computer Science 2320, Springer-Verlag, 2002, pp. 1-12.

[9] P. Junod, Obfuscator-LLVM, 2013. url: http://www.o-llvm.org/.

[10] D. Lambić and M. Živković, “Comparison of Random S-Box Generation Methods,” Publ. de L’Institut Mathématique, Nouvelle série, tome 93 (107) , 2013, pp. 109-115.

[11] GlobalPlatform Card Specification, Version 2.2.1, Public Release, January 2011. url: http://www.globalplatform.org/specificationscard.asp