Proceedings of the 2016 Winter Simulation Conference
T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, eds.

A METHOD TO AVOID SMARTPHONE MEMORY ERRORS IMPACTING ENCRYPTION KEYS

Jianing Zhao
Peter Kemper

Department of Computer Sciences
College of William and Mary
Williamsburg, VA 23185, USA

ABSTRACT

Encryption is used as the method of choice to control access to sensitive data on a smartphone by systems such as CleanOS. We present a simulation study to demonstrate the potential damaging effect that memory errors can have on encrypted data if errors corrupt encryption keys. We show how simple algorithmic strategies to detect and correct a faulty key can marginalize the risk of such errors.

1 INTRODUCTION

The proliferation of smartphones and their use in everyday life for personal and business purposes has led to the situation that phones store a significant amount of sensitive data. Therefore much research has gone into the identification, tracking and protection of sensitive information. The CleanOS system Tang et al. (2012), for example, relies on encryption and the remote storage of keys in the cloud to control access to sensitive data and to prevent decryption of locally stored, encrypted data in case of theft. However, if memory errors corrupt encryption keys, encrypted data may become accidentally inaccessible. As main memory in most smartphones is not protected against memory errors, holding encryption keys only in main memory over time as done in CleanOS is susceptible to memory errors. Zhao and Kemper (2016) show some preliminary findings on ways memory errors can lead to loss of sensitive data under the CleanOS architecture.

In this paper, we further extend results from Zhao and Kemper (2016) by simulating the effect of algorithmic strategies to prevent memory errors from affecting encryption keys. The research question we investigate is: **To what extent can software solutions alleviate the possible destructive effects of memory errors on encryption keys for the encryption of sensitive data in a key escrow system such as CleanOS?** We consider the following threat model: We assume that content in main memory can be corrupted (single or multiple bits) at any moment in time due to memory errors. Possible root causes are software errors as well as hardware failures. Programming errors in kernel or application code can accidentally overwrite content in main memory. Programming languages such as C are infamous for the numerous possibilities to make mistakes with pointer arithmetic, dangling pointers and so forth (van der Veen et al. 2012). Single or multiple bit errors can also be the result of hardware errors on unprotected DRAM. ECC protection for DRAM is technically possible, but comes at a price in terms of increased production costs, reduced capacity and increased energy consumption. So for most smartphones, it is currently common that DRAM is not equipped with ECC. Although the probability of a memory error is small, it is still quite possible that it happens and a stored key may be corrupted by some hardware or programming error.

We investigate existing solutions in related work in Section 2. In Section 3, we discuss specific solutions to make the key management in encryption-based systems like CleanOS resilient to memory errors. We

978-1-5090-4486-3/16/$31.00 ©2016 IEEE

come up with a simple algorithm and perform a simulation study to evaluate its effectiveness in Section 4 and its performance in Section 5.

2 RELATED WORK

Hardware faults are one source of memory errors. For DRAM used in large scale data centers, studies by Hwang, Stefanovici, and Schroeder (2012) and Schroeder, Pinheiro, and Weber (2009) show that there is a rising rate of memory error occurrences despite efforts in quality control of DRAM production and error-tolerance mechanisms. There are many causes for memory errors such as environmental factors as shown in Kim et al. (2014), Khan et al. (2014), Liu et al. (2013). For mobile memory, we did not find corresponding numbers being published. However, high density and low cost design are arguments to justify the assumption that DRAM in mobile devices is less reliable than PC DRAMs at least for the low-end of the market.

A second source of memory errors result from programming errors such as buffer overflow and dangling pointers in software developed in languages such as C and C++.

2.1 Existing Hardware-based Solutions

A number of common memory error detection and correction techniques exists such as Parity, SEC-DED, Chipkill (Dell 1997) and DEC-TED that all rely on some variant of error correction codes (ECC) to handle a small number of bits or chip errors. Techniques such as RAIM or Mirroring to correct errors towards the size of a module come with high cost in terms of added capacity or logic. See Luo et al. (2014) for an excellent, brief overview. According to Malladi et al. (2012), mobile DRAM such as LPDDR2 has x16 width compared to x4 or x8 width for the common DDR3. This creates a challenge because original ECC code is not designed for x16 width DRAM.

Nair, Kim, and Qureshi (2013) describe a method to deal with high memory error rates caused by higher density DRAM technology that increases capacity per area. They use a fault map to store the location of bad memory cells and then operate with a copy of these locations. Although technology such as ECC is an effective way to detect and correct hardware-caused memory errors, it is currently not employed in DRAM for mobile devices. So we conclude that a robust mobile OS can not rely on the presence of a strong hardware layer to handle memory errors.

2.2 Existing Software Solutions

Pattabiraman, Grover, and Zorn (2008) propose an OS kernel extension to protect critical data from memory errors in their Samurai system. Critical data is stored multiple times to allow for detection of an error as differences between redundant copies and its subsequent correction by majority voting to determine the correct content. Samurai is implemented at the OS level and little change is required for applications besides the need to tag critical fields.

Yoon and Erez (2010) propose a virtual ECC that maps redundant information to a different memory location where it is visible to the program. The benefit of this method is the flexibility to choose ECC protection methods, but the memory management system needs to be modified.

All these systems need to establish some form of redundancy to allow for the recovery of errors. In the following we explore a design that benefits from the fact that keys in a system like CleanOS have a trustworthy copy of a key that can be retrieved when needed.

3 A DESIGN TO PROTECT SDOS IN CLEANOS AGAINST MEMORY ERRORS

The basic idea of the CleanOS system is to encrypt sensitive data already in main memory and use encryption keys that only reside temporarily in main memory and are obtained from a cloud service as needed. A Sensitive Data Object (SDO) is defined in Tang et al. (2012) to represent and administer sensitive data.

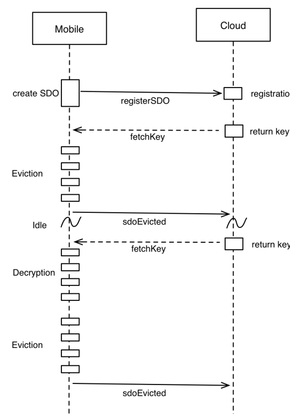


Figure 1: CleanOS communication between phone and cloud for the one time, initial registration of an SDO, encryption and key eviction (top part) and the regular operation sequence of fetching the key, data decryption, data access, data encryption and key eviction (bottom).

An SDO has a key for encryption and a descriptor that is necessary to obtain the key from the cloud service. The analysis of CleanOS in Zhao and Kemper (2016) revealed vulnerabilities for memory errors to escalate into data loss for an SDO and Figure 1 in Zhao and Kemper (2016) shows the communication between mobile and cloud. The time window between fetching the SDO key from the remote service and using it for encryption or decryption gives an opportunity to corrupt the key and to cause data loss. A different vulnerability results from corrupting the SDO descriptor that leaves the CleanOS system unable to request the key for an SDO from the remote service. In conclusion, the SDO (or bucket (Krawczyk and Eronen 2010)) key and the SDO descriptor are pieces of data that need protection against memory errors. SDO key and descriptor are both small in size and present in same quantities. However there are important differences: the key information naturally has a trusted, valid copy in the cloud; its eviction on the phone requires secure deletion of the key. Having multiple local copies is contrary to the CleanOS design's policy of having tight control over the SDO by controlling access to the key. The SDO descriptor on the contrary is not controlled by the cloud service, it is not particularly secret and having multiple local copies is acceptable. Although a trusted, valid copy of the SDO descriptor exists on the cloud side, there is no functionality on the phone side to obtain this copy. For brevity of presentation, we focus on the key and only argue about the descriptor if it requires a separate discussion. To achieve the goal of protecting the keys, the following two components are crucial.

Error detection For a software-based solution that allows for error detection and correction, we need to establish a boolean function detect(x) that returns true iff x is invalid and a function correct(x) that returns the correct value for x. Function detect(x) can be established in different ways which all rely on some form of redundant information to support a comparison operation. One way is to store copies of x and perform a direct comparison. The other way is to check if the hash value of x changes over time. The latter saves on space but more importantly it prevents keys from being stored elsewhere.

Error correction As discussed in the section on related work, a variety of techniques exist for recovering data after a memory error. However, error correction is straightforward if a valid copy can be requested from a trusted source as it is the case for the SDO key and the fetchKey operation that obtains a valid copy from the cloud service. Since key eviction and refetching is part of the normal operational routine, refetching a key is the natural choice of error correction for the SDO key. The induced extra overhead for network communication corresponds directly to the frequency of memory errors which is expected to be low.

The SDO descriptor has different characteristics as it can not be recovered from the cloud services but it also need not be evicted or protected with encryption, retaining several copies for error correction

is acceptable. This makes the SDO descriptor a good example for a piece of critical data protected with Samurai or a reimplementation of the Samurai concept.

3.1 Preventing Data Loss in CleanOS

We propose the simple algorithm 1(Keyguard) which uses error detection and correction mechanisms to prevent that vulnerabilities result in a data loss in the CleanOS system. For simplicity of presentation, we consider one SDO, one key and one descriptor (arbitrary but fixed). We consider the key in lieu of the SDO key or the bucket key because the only difference is that for the bucket key one needs to obtain the SDO key and compute the bucket key. We begin with vulnerabilities that result from corrupting the key. As the cloud service provides a trusted, valid copy on demand anyway, our error correction is simply based on fetching the key. For error detection, we use a hash code. We attach the hash value to the key such that key and hash value are stored and retrieved together from the remote server.

Algorithm 1 describes the enhanced basic encryption and decryption routines to include error correction and detection. We assume that the key, its hash value, its descriptor, and the sensitive data are accessible through corresponding variables named key, hash value, descriptor and data. We also assume the existence of a hash function named HASH. The CleanOS function to fetch the key is named FETCHKEY and the function actually used for encryption is named AESENCRYPT. Function DETECT implements the error detection based on hashing. Function CORRECT implements the error correction by refetching the key from the cloud. Function DETECTANDCORRECT performs an error correction as often as necessary to obtain a valid key. The handling of the encryption and decryption process are very similar with corresponding observations towards correctness, limitations, termination and performance such that we discuss only the encryption part in the following.

The two main ideas for the data encryption in function ENCRYPT are 1) to ensure that a key is valid at the beginning of an encryption operation and to recover it if necessary (line 12) and 2) to check the key again at the end of the encryption operation (line 14) and before overwriting thus destroying the clear text data (line 15). Line 14 is the last moment to redo the encryption if the key turns out to be invalid. The rationale is that if the key is valid before and after the encryption operation, then it was also valid during the encryption operation. The conclusion is only true if the possibility of two successive changes to the key that balance out is negligible, which we assume. With this assumption, we can argue that ENCRYPT satisfies the key consistency condition. Note that this algorithm is limited to protecting the key and does not consider memory errors that corrupt data or descriptor. With regard to termination, function DETECTANDCORRECT may lead to an infinite loop. In practice, one would limit the number of attempts by a threshold value and exit with an error code if the valid key can not be obtained within given number of attempts. In terms of network communication overhead, the number of additional FETCHKEY operations that imply communication with the cloud service is directly proportional with the number of memory errors that corrupt the key such that the overhead is assumed be marginal. In terms of performance overhead, the hash function and comparison in function DETECT is computationally inexpensive and the operation to fetch the key is the same as for the regular CleanOS operation. Repeatedly encrypting the data in line 13 may be computationally expensive but the number of iterations is the same as the number of memory errors that corrupt the key in that time frame such that an actual additional iteration with line 13 is rare.

There are three further scenarios worth considering. If the key is right, but the hash value is corrupted by memory errors, we can not distinguish this from a corrupted key and hence, we refetch the key-hash value pair and check the hash value again. This case is at most as likely as the key corruption and its impact is moderate as it results in an unnecessary fetch key operation. Another scenario is a hash collision. The impact of a hash collision for a key and a corrupted key would make the DETECT method fail to recognize a corrupted key which will then lead to a loss of data for the ENCRYPT function. The frequency of a hash collision depends on the chosen hash function and length of hash values. For instance, SHA1(Eastlake and Jones 2001) is a frequently applied cryptographic hash function and it has 160 bit values. In our design, we assume with a reasonable length of hash values, such that the probability of a collision is negligible.

Another extremely unlikely event is that of two memory errors that corrupt key and hash value such that HASH(key) and hash value match with the consequence of a faulty result of the DETECT method. In summary, the additional error detection and correction functions in Algorithm 1 suffice to prevent the key from corrupting in almost all possible scenarios. This addresses vulnerabilities I and III listed in Zhao and Kemper (2016)

Algorithm 1 Basic encryption/decryption operation with error detection and correction.

```
 1: function DETECT(key)
 2:     return(hashvalue != HASH(key))
 3: function CORRECT(descriptor)
 4:     (key,hashvalue) ← FETCHKEY(descriptor)
 5: function DETECTANDCORRECT(key,descriptor)
 6:     while DETECT(key) do
 7:         CORRECT(descriptor)
 8: function ENCRYPT(data)
 9:     if key not in cache then
10:         (key,hashvalue) ← FETCHKEY(descriptor)
11:     repeat
12:         DETECTANDCORRECT(key,descriptor)
13:         tmp ← AESENCRYPT(data,key)
14:     until false == DETECT(key)
15:     data ← tmp
16: function DECRYPT(data)
17:     if key not in cache then
18:         (key,hashvalue) ← FETCHKEY(descriptor)
19:     repeat
20:         DETECTANDCORRECT(key,descriptor)
21:         tmp ← AESDECRYPT(data,key)
22:     until false == DETECT(key)
23:     data ← tmp
```

Key generation So far we focused on the fetching and using the key for encryption or decryption. With regard to the generation of keys, a key can be generated either locally in the device or remotely in the trusted cloud. The CleanOS design suggests a local generation and sending the key to the cloud upon registration. This allows for an opportunity to register a corrupted key which is not satisfying as such but does not lead to an inconsistency between local and remote key. Since we suggest to attach the hash value to the key and register both with the remote server, the correctness of the key can be easily checked on both sides in the key registration process.

Protection of descriptor We recognized that the descriptor is essential to fetch a key from the cloud. Consequently, a corrupted descriptor prohibits obtaining the key for decryption, which in turn prohibits decryption of encrypted sensitive data. The descriptor itself is not part of the sensitive data and encryption. It naturally occurs in multiple locations such as the metadata of each SDO, individual pieces of sensitive data, for example the entries in SQLite database columns that are added to accompany columns with sensitive data. While the metadata that includes the descriptor information is securely stored on persistent storage by the CleanOS system, we did not find a way to identify a corresponding meta entry if the descriptor is not known (corrupted). The descriptor is a good example of the data the Samurai OS extension envisions

as ”critical”. If the Samurai concept can be refined to apply to TaintDroid storage, the descriptor should be protected in all of its locations.

Protection of data In the above, we discussed memory errors to keys because keys are more sensitive to memory errors than data. However, it is quite possible that data are corrupted by memory errors. When we changed one bit of sensitive data and used AES encryption to encrypt and decrypt the data, we observed that the bit that was changed was lost after decryption but the other bits of sensitive data were correct. On the contrary, if we change one bit of the key, then after decryption we observed that the whole sensitive data is lost. If we apply the aforementioned method used to protect keys to sensitive data, more extra resources are needed. For example, if the sensitive data is big, it takes more time and uses more space to detect errors in sensitive data and correct them.

4 RELIABILITY EVALUATION

In this section we present findings from a simulation study to demonstrate how vulnerable the handling of keys in a CleanOS system is to memory errors and to what extent the suggested improvements in Algorithm 1 can withstand injected memory errors that destroy the key.

We first use Mobius (Deavours et al. 2002) to simulate the CleanOS and Keyguard , then we implement basic usage scenarios in C and use a concurrent thread for fault injections to evaluate our approach as a proof of principle. Both simulation and C implementation is based on the CleanOS source code analysis.

4.1 Evaluation using Mobius

We use Mobius (Deavours et al. 2002) to establish and simulate the Keyguard algorithm using SAN models. We use one submodel for Keyguard and one submodel for fault injection as shown in Figure 2 and Figure 3. The parameters used in the experiment are shown in Table 1. We also evaluate a simpler version of the model in Figure 2 that only measures failures but does not refetch a key to measure the effect of memory errors on the basic CleanOS system without Keyguard. We use an exponential distribution to model the time between memory faults, denoted as fault injection rate. The mean time between faults ranges between 0.87 and 2.4 hours. For the other activities, we assume a deterministic distribution for simplicity and use published average values from Tang et al. (2012). Idle time denotes the time between phases of active usage of an SDO, its parameter value was estimated from measurements of app usage for two volunteers in a period of two weeks. Measurements were recorded with the QualityTime app.

In Figure 2, the activities named keyfetch, decryptionRefetch, encryptionRefetch, and eviction have a duration that is modeled with the network communication delay in Table 1. The duration of other activities follows corresponding distributions shown in Table 1. The whole system is described in previous sections and we do not describe further details of the Mobius model for space limitations. We conducted simulations for different rates of fault injections, namely for a mean time between faults in [0.87,2.4] such that for an exponential distribution, $\lambda \in [0.4, 2.0]$. All simulations were performed with the Mobius simulator for a terminating simulation, confidence level setting of 95% and confidence interval setting of 0.1.

In Figure 4, we observe several measures as a function of the fault injection rate. Red lines show results for an average idle time of 6 minutes, blue lines for an idle time of 12 minutes between active phases for an SDO. A longer idle time reduces the number of times the system can go through a decryption, SDO active usage, SDO inactive timeout, encryption, key eviction cycle (a CleanOS cycle) in a fixed time horizon of 5000 hours. For sanity checks, Figure 4b shows the total number of cycles (correct or failed) being independent of the fault injection rate and that shorter idle times yield more CleanOS cycles (red line vs blue line). Figure 4a shows that the basic CleanOS model without key protection experiences failures that increase with the fault injection rate and that the more CleanOS cycles are performed the higher the number of failures (difference between red line and blue line). Figure 4c shows results for the CleanOS model with Keyguard. Between 230-1400 (180-600) memory errors that cause key corruption are successfully detected for 6 minutes (12 minutes) idle time and corrected with refetch operations. The key

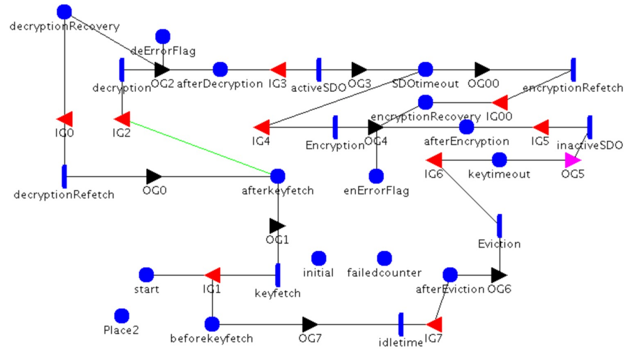
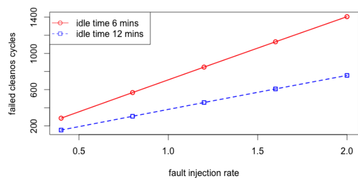


Figure 2: Mobius: keyguard model.

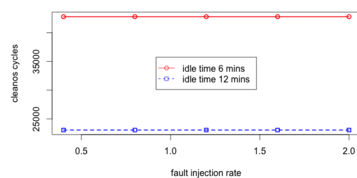
Figure 3: Mobius: fault injection model.

Table 1: parameters in the model.

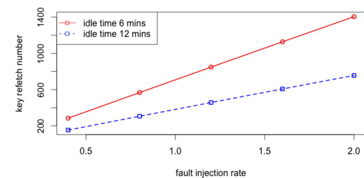
parameter	**distribution**	**value**	**source**
decryption, encryption delay	Deterministic	2ms	(Tang et al. 2012)
network communication delay	Deterministic	300 ms	(Tang et al. 2012)
fault injection rate	Exponential	Mean:0.87-2.4 per hr	(Schroeder et al. 2009)
SDO active, SDO inactive timeout	Deterministic	60 s	(Tang et al. 2012)
idle time	Deterministic	0.2 hr	qualitytime used by two voulunteers
experiment time interval	-	5000 hrs	assume a smartphone is used 14 hours daily for a year



(a) Failed cycles without Keyguard.



(b) Total cycles.



(c) Keyguard key refetch number.

Figure 4: Mobius simulation result.

refetch number is basically the same as the failed CleanOS cycles in Figure 4a as all failures are detected and a single refetch operation apparently suffices in most cases. For the chosen model distributions and parameter values, we observe from our simulation that failures are expected in the order of hundreds per year and that Keyguard's overhead is basically one additional key fetch operation per failure. We can see the corrupt keys in the daily usage scenario is in the order of 10^3 for entire memory. In the simulation, we assume all fault injections target keys, although in reality memory errors spread across memory.

4.2 Evaluation using Two Threads in C

We also implemented two usage scenarios in C that follow the operational model for encryption and decryption in the CleanOS system and use AES encryption code. The first usage scenario describes the regular access to sensitive data in an existing SDO and its eviction; it consists of the following sequence of steps:

Usage Scenario I

1. The first access to SDO data triggers key fetching from the trusted location. Perform decryption and store a local copy of key in cache. Let k denote the size of SDO.
2. Perform n-1 other data accesses that require decryption and use the cached copy of the key.
3. After a delay d, perform an idle eviction: encrypt all n pieces of sensitive data that is currently in clear text form and delete the local copy of the key.

This scenario is then repeatedly executed over time. As we can recognize from this scenario, caching the key creates an extended period of time where a memory error can corrupt the key. Therefore, we also look into a second scenario in that keys are not cached:

Usage Scenario II

1. Any access to encrypted SDO data triggers key fetching rom a trusted location. Perform decryption of data.
2. After delay d and idle eviction, key is fetched from a trusted location. Perform encryption of data.

This scenario is then repeatedly executed over time. We set the idle time to zero to create a stress test scenario where a fault injection is basically effective all the time. We measure failures by counting failed scenarios. We consider a scenario failed if the encrypted data is lost at the end of the scenario, i.e., it can not be decrypted with a valid key anymore. We confirmed with a separate and independent test that a memory error that hits the data leads only to a partial data loss of corresponding size (only the corrupted bytes can not be successfully decrypted again). However that test also showed that a single bit error on the key escalates into a complete loss of the encrypted data.

Parameters for the first scenario are 1) n, the number of data items decrypted and encrypted, 2) k, the average size of the data items as this contributes to the execution time for encryption and decryption, 3) the timeout delay d to start the idle eviction. For the second scenario it is just the size of the data items and the time delay. For the fault injection (FI) model, faults are injected after random delays such that we can measure an average rate of fault injections per time unit. We decided to implement this model in C code and with pthreads. We use one thread for the implementation of the CleanOS usage scenario and one thread for the FI model. The key is a shared data object with unsynchronized access by both threads which establishes a race condition. The FI thread overwrites the content of the key, while the CleanOS thread reads the content of the key for encryption/decryption operations and overwrites it when fetching a key. We do not use a cloud service, a trusted location is a memory location that is not subject to fault injections. We decided to use an implementation in C in order to stay close to the existing CleanOS implementation that is implemented in C and uses the same code for encryption. Thanks to the multithreading, the code in the CleanOS thread has no information on the fault injections. The thread scheduler influences the timing of fault injections. In order to avoid scheduling artifacts, we only use calls to thread method "sleep" for

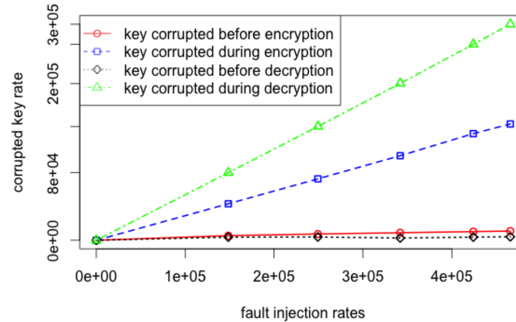


Figure 5: CleanOS usage scenario II with k=16 bytes: number of successful fault injections per second into the key differentiated by occurrence before/during encryption/decryption as a function of the total fault injection rate.

delays on the FI thread but not on the CleanOS thread. We use a random delay with a fixed average value that is experimentally configured to deliver the highest possible FI rate. We then throttle the actual FI rate with the help of a second random variable to randomly decide to overwrite the key value or not. Varying the probability for an overwrite in the second random variable does not influence the thread scheduler such that we can vary the actual FI rate without modifying the amount of CPU time given for each thread. We trace the FI and CleanOS thread activity in log files with time stamps and we carefully checked that both threads get executed, that FI strikes the CleanOS thread in all possible stages of execution, and that the time interval we evaluate for measurements is one where both threads perform.

Figure 5 shows that the fault injection is successfully placing errors into the keys before and during encryption and decryption for the repeated execution of usage scenario II without key protection. The number of successful injections is proportional to the time spend in each of the phases. One can clearly see that decryption takes much longer time than encryption while the phases between fetching the key and using the key are very short but it is still possible to inject faults. We exercised similar sanity checks for larger values of k with corresponding results.

A successful fault injection corrupts the key and makes a usage scenario fail. Figure 6 shows how the rate of successful executions of usage scenario II deteriorates if there is no protection of the key and one increases the fault injection rate. The experiment shows that a memory error that hits the key can escalate into data loss for an SDO. Note that the size of an SDO can be substantial, in Tang et al. (2012), coarse default SDOs such as a *Password* SDO for all passwords and a *SSL* SDO for all objects read from SSL connections are established which makes such a data loss significant.

Figure 7 shows our main result for parameter settings n=10, k=16 bytes, d=0; we vary the rate of fault injections per second over a series of experiments and observe the average number of failed scenarios (loss of sensitive data) per second. The red line shows the number of failures for the first usage scenario that models the CleanOS behavior most closely. There is no protection against memory errors and the results show the steepest slope and highest values of all experiments. The black line relates to the second usage scenario for a CleanOS variant with no caching of keys and no protection against memory errors. The failure rate drops significantly which is caused by the minimal time between fetching the key and using it for encryption or decryption. Obviously the price for this is an increase in network communication. The green and blue lines that coincide and remain constantly at zero are the results of using the key protection of Algorithm 1 (Keyguard) in experiments for the first and second usage scenario. The experiment supports our claim that Keyguard reliably detects and corrects key corruptions and thus effectively prevents failures.

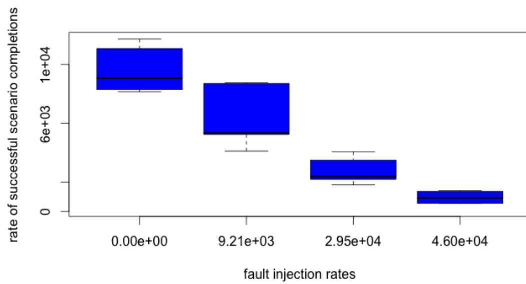


Figure 6: Rate of successful eviction cycles for the CleanOS usage scenario II as a function of fault injection rates.

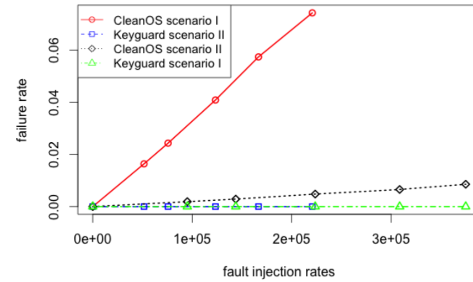


Figure 7: Observed failure rates of usage scenarios I and II with key protection (keyguard) and without (CleanOS) as a function of fault injection rate (injections/second).

5 PERFORMANCE EVALUATION

From Figure 7 one can clearly see that fetching keys instead of caching keys reduces the chances of a memory error to corrupt the key significantly (black line vs red line in figure). However, caching the key implies that the number of fetch key operations $\#fetchkey \propto \#evictions$ as there is one fetch key operation per idle eviction operation. Without caching, communication costs for the number of fetch key operations rise dramatically to $\#fetchkey \propto (\#encryption + \#decryption)$ as each encryption and each decryption operation triggers a fetch key operation of its own. This is why the actual CleanOS design emphasizes caching for SDO keys and bucket keys and follows usage scenario I. The Keyguard protection mechanism in Algorithm 1 supports caching but it also increases the potential number of fetch key operations. The number of fetch key operations is $\#fetchkey \propto (\#evictions + \#memoryfaults)$ as each successful corruption of the key is detected and corrected with the help of an additional fetch key operation. This proportion can be also seen in the experimental results. We measured the number of fetch key operations for the usage scenario II and a varying number of fault injections. Figure 8 shows a series of box plots for a set of experiments with increasing fault injection rates. As the number of memory faults that corrupt the key is expected to be low over time, so is the additional overhead on the network communication.

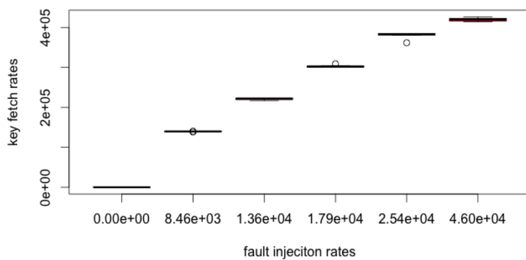


Figure 8: Boxplot with rates of keyfetch operations (calls/second) for a usage scenario II with Keyguard in response to different fault injection rates.

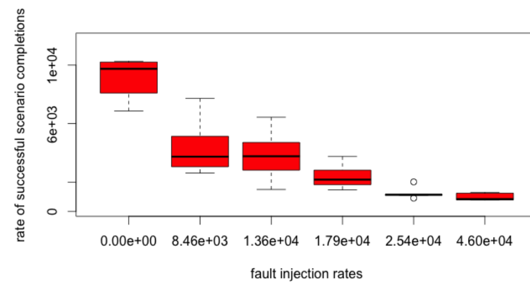


Figure 9: Boxplot with rates of successful executions of usage scenario II with Keyguard in response to different fault injection rates.

An increase in the number of fetch key operation drags on the overall performance. Figure 9 shows the processing rate for usage scenario II with Keyguard for different fault injection rates. When the fault injection rate increases, the successful cleanos rate decreases because we need to refetch the keys more

often. The time cost by a successful cleanos cycle increases as the refetch keys are more often caused by increasing fault injection rates. Note that the fault injection rates are not meant to be realistic but chosen to demonstrate the effect. From Figure 5, we can recognize that encryption and especially decryption operations are computationally expensive. Options to improve on this are known. As noted in Tomoiaga and Stratulat (2010), Gleeson, Rajan, and Saini (2014), it is possible to use GPU to encrypt and decrypt sensitive data in order to reduce the workload of CPU. Furthermore, Suh et al. (2003) use one-time-pad encryption to reduce decryption latency. In the AES algorithm under CBC mode, the decryption starts only after all data is read completely. By applying Suh's method, the decryption can start after the first chunk of data is read, hence, data access and decryption can overlap.

6 CONCLUSION

In this paper, we did a simulation study of CleanOS to see how memory errors can affect its operation and a simple algorithm to deal with memory errors in such case. Based on several scenarios identified in Zhao and Kemper (2016) where corruption of a key that is used for encryption can lead to loss of sensitive data, we discuss possible solutions for error detection and correction and present a specific algorithm that naturally fits into the overall setting as a purely software-based solution. The algorithm imposes marginal overhead in the network communication that is proportional to the number of memory errors that corrupt the local copy of the encryption key. We conduct two simulation studies, one based on stochastic model in Mobius, the other closer to running actual implementation code with a thread-based fault injection to evaluate the risk of data loss with and without our approach. We believe that our suggested strategy naturally generalizes to other architectures such as Keypad that also rely on encryption and remote storage and monitoring with the help of encryption keys.

ACKNOWLEDGMENTS

Thanks to Roxana Geambasu for sharing the CleanOS source code with us.

REFERENCES

Deavours, D. D., G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. 2002, October. "The MöBius Framework and Its Implementation". *IEEE Transactions on Software Engineering* 28 (10): 956–969.

Dell, T. J. 1997. "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory".

Eastlake,3rd, D. and Jones, P. 2001. "US Secure Hash Algorithm 1 (SHA1)".

Gleeson, J., S. Rajan, and V. Saini. 2014. "GPU Encrypt: AES Encryption on Mobile Devices".

Hwang, A. A., I. A. Stefanovici, and B. Schroeder. 2012, March. "Cosmic Rays Don'T Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design". *ACM SIGPLAN Notices* 47 (4): 111–122.

Khan, S., D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu. 2014, June. "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study". *ACM SIGMETRICS Performance Evaluation Review* 42 (1): 519–532.

Kim, Y., R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. 2014. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 361–372. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc.

H. Krawczyk and P. Eronen 2010, May. "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)". https://tools.ietf.org/html/rfc5869.

Liu, J., B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. 2013, June. "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms". *ACM SIGARCH Computer Architecture News* 41 (3): 60–71.

Luo, Y., S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. 2014. "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory". In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, 467–478. Washington, DC, USA: Institute of Electrical and Electronics Engineers, Inc.

Malladi, K. T., B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. 2012, June. "Towards Energy-proportional Datacenter Memory with Mobile DRAM". *ACM SIGARCH Computer Architecture News* 40 (3): 37–48.

Nair, P. J., D.-H. Kim, and M. K. Qureshi. 2013. "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates". *ACM SIGARCH Computer Architecture News* 41 (3): 72–83.

Pattabiraman, K., V. Grover, and B. G. Zorn. 2008. "Samurai: Protecting Critical Data in Unsafe Languages". In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, 219–232. New York, NY, USA: ACM.

Schroeder, B., E. Pinheiro, and W.-D. Weber. 2009. "DRAM Errors in the Wild: A Large-scale Field Study". In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, 193–204. New York, NY, USA: ACM.

Suh, G. E., D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas. 2003. "Efficient Memory Integrity Verification and Encryption for Secure Processors". In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 339–. Washington, DC, USA: Institute of Electrical and Electronics Engineers, Inc.

Tang, Y., P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. 2012. "CleanOS: Limiting Mobile Data Exposure with Idle Eviction". In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 77–91. Berkeley, CA, USA: USENIX Association.

Tomoiaga, R., and M. Stratulat. 2010. "AES Performance Analysis on Several Programming Environments, Operating Systems or Computational Platforms". In *2010 Fifth International Conference on Systems and Networks Communications*, 172–176. Institute of Electrical and Electronics Engineers, Inc.

van der Veen, V., N. dutt Sharma, L. Cavallaro, and H. Bos. 2012. "Memory Errors: The Past, the Present, and the Future". In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, 86–106. Berlin, Heidelberg: Springer-Verlag.

Yoon, D. H., and M. Erez. 2010, March. "Virtualized and Flexible ECC for Main Memory". *ACM SIGPLAN Notices* 45 (3): 397–408.

Zhao, J., and P. Kemper. 2016. "Protecting Encryption Keys in Mobile Systems Against Memory Errors". In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, 224–227. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

AUTHOR BIOGRAPHIES

JIANING ZHAO is a PhD candidate of Computer Science at the College of William and Mary. His research interests lie in simulation modeling, data mining, causal inference especially in social good. His email address is jzhao@cs.wm.edu.

PETER KEMPER is an Associate Professor in the Department of Computer Science at the College of William and Mary (previously TU Dortmund and TU Dresden, Germany). His research interests include modeling techniques and tools for performance, performability and dependability analysis of systems. He contributed to analysis techniques for the numerical analysis of Markov chains, model checking stochastic models, techniques for simulation optimization. His email address is kemper@cs.wm.edu.