Poster: Partitioning Garbage Collection Between the Secure and Normal Worlds for Trusted Applications

Harishankar Vishwanathan, Chang Min Park, Sidharth Kumar Mishra,
Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
University at Buffalo, The State University of New York
Buffalo, New York
{harishan,cpark22,smishra9,kdantu,stevko,lziarek}@buffalo.edu

ABSTRACT

Trusted Applications (TAs) written for Trusted Execution Environments (TEEs) using ARM TrustZone are currently written in C; there is limited support for higher-level languages. This leads to common manual memory management problems like buffer overflow and use-after-free. Higher-level languages, which have managed runtimes, allow for automated memory management, the benefits of which are widely accepted. To allow for automated memory management of TAs, we need to have a runtime that handles allocation and garbage collection (GC). However, having the entire allocator and GC in the secure world would increase the Trusted Computing Base (TCB) of the secure world. We propose TrustGC, a mechanism to partition garbage collection and allocation between the secure world and the normal world. TrustGC allows for automated memory management of TAs by leveraging the help of a GC partly running in the normal world.

CCS CONCEPTS

• **Security and privacy** → **Systems security; Trusted computing**; • **Software and its engineering** → **Memory management; Garbage collection**.

ACM Reference Format:

Harishankar Vishwanathan, Chang Min Park, Sidharth Kumar Mishra, and Karthik Dantu, Steven Y. Ko, Lukasz Ziarek. 2019. Poster: Partitioning Garbage Collection Between the Secure and Normal Worlds for Trusted Applications. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*, June 17–21, 2019, Seoul, Republic of Korea. ACM, New York, NY, USA, 2 pages. <https://doi.org/10.1145/3307334.3328650>

1 INTRODUCTION

Modern computing systems depend on a large TCB: the OS kernel, privileged services, and libraries. Applications that rely on such a large TCB are naturally vulnerable to security threats inherent in a large TCB. ARM TrustZone is a hardware mechanism in ARM application processors that allows for the creation of two protection domains: the secure world and the normal world. At a given point, software executed by an ARM processor runs exclusively in the

Figure 1: Partitioning Garbage Collection.

secure state or the non-secure state. This allows for enforcing a Trusted Execution Environment (TEE) in which Trusted Applications (TAs) can execute without interference of an untrusted OS. TAs are typically used to build security services related to DRM, mobile payments, secure banking, multi-factor authentication, device reset protection, secure PIN and fingerprint processing, etc. Pinto et al. [2] present an in-depth survey of TrustZone-enabled systems for TEEs.

TrustZone TAs are security-critical applications which are part of the TCB of the system. They are currently written in C, and there is limited support for higher-level languages. It is well known that applications written in C have common memory management problems associated with them including buffer overflows and user-after-free bugs.

There have been many vulnerabilities regarding TrustZone, many of which are related to memory management [2]. Developing TAs in higher-level languages with managed runtimes would help alleviate the developer's burden for performing memory safety checks. However, having a managed runtime for TAs in the secure world would significantly increase the TCB. Our approach to solving this problem involves partitioning the garbage collection and allocation mechanism between the secure world and the normal world, and performing a (major) portion of the garbage collection in the normal world. This way, a TA can benefit from automated memory management while not having to rely on a large TCB to do the task. We note that a broader goal is to implement a runtime for TAs, but in this poster, we focus on the GC and the allocator. Figure 1 shows a basic overview of the system.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6661-8/19/06.

<https://doi.org/10.1145/3307334.3328650>

2 RELATED WORK

Several runtime systems have been developed previously in the context of TrustZone and the TEE infrastructure. TLR [4] is a small runtime engine capable of interpreting .NET managed code inside a trusted environment using the .NET Micro Framework. TLR allows application developers to factor out security-sensitive app logic to run in a trusted environment. Alternatively, the work by Rubinov et al. [3] provides an automated partitioning framework to generate code segments to be deployed as TEE commands of a TA by using taint analysis to track propagation of sensitive data in an Android application. TrustShadow [1] protects legacy applications from the untrusted OSes running them, without having developers re-engineer their applications. It creates a TEE for legacy applications and uses a lightweight runtime system to coordinate communications between applications and untrusted OSes. While all of these solutions focus on providing secure execution of normal world applications, our focus is entirely on TAs running in the secure world.

3 PARTITIONING GARBAGE COLLECTION

3.1 Mark-and-Sweep Collection

Mark-and-sweep is a common garbage collection mechanism which several runtime systems build on. In this section, we present a brief overview of the mark-and-sweep GC, which we use as a motivating example for partitioning garbage collection. The header of an allocated block contains the data structures necessary for allocation and collection: a bit to identify whether a block is allocated or not, the size of the block, a pointer to other blocks, and a mark bit. We can view memory as a directed graph, with each allocated heap block as a node in the graph, and each pointer as an edge on the graph, and root nodes as pointers that point to the heap but are not in the heap themselves. The idea behind mark-and-sweep is that a node is *reachable* if there is a path from any root node to that node, and all non-reachable nodes can be garbage collected as they cannot be needed by the application. The mark operation starts at the roots of the graph, and sets the mark bit on each reachable block. The sweep operation then scans all blocks and frees blocks that are not marked. Given this algorithm, we can think about presenting the normal world with a view of the TA's heap to allow garbage collection operations on it. We consider two solutions which are discussed below.

3.2 Partitioning Approaches

A strawman solution: The first solution is to have the secure world encrypt the contents of the TA's heap. This addresses the issue of not allowing the normal world GC to view the security-critical contents of the TA's memory. Whenever the GC algorithm needs to run (say, if there is not enough memory to be allocated), a call is made to the normal world GC. The GC algorithm runs completely in the normal world over encrypted data. In order for this to work, we need to maintain the meaningfulness of the pointers and the metadata when we encrypt the secure world memory. The TCB size is relatively small: only the encryption, decryption, and some bookkeeping operations would be needed in the secure world.

A second design: The strawman, however, gives the normal world write-access to the TA's memory, exposing it to heap overflow, heap spraying, and other attacks. To address these issues, we propose a second design. Here, we have a shared memory region between the secure world and the normal world. The allocator in the secure world writes unencrypted metadata and pointers to the shared memory region along with every allocation in the secure world, and zeroes out bits for non-pointer data. The GC in the normal world then runs on this copy of the TA's heap. Once it completes, the secure world looks-up this shared memory space and essentially replicates the sweeping phase to actually collect garbage memory. Because we do not allow the GC write-access to the TA's memory, we protect it from the previously mentioned attacks.

Any GC algorithm could run in the normal world for both the designs in theory. However, we cannot protect the GC from collecting data that it should not (say, by arbitrarily clearing the mark bit on a reachable block). In other words, we cannot prevent a GC running in a potentially compromised OS from being malicious. However, in the second design, there could be some mechanisms implemented in the secure world to check and guard against the operations performed by the normal world GC before actually collecting memory. For instance, if the allocator in the secure world allocates dummy objects and pointers, the GC algorithm can still function, but the secure world partition of the GC can use that to check if the GC is performing correctly. We note that the TCB size would be larger in the second design, depending on what operations need to be performed in the secure world.

4 FUTURE WORK

We plan to explore other ideas and build on the two mentioned above. We plan to explore the interfaces for different GC algorithms, explore specific threat models, and study what guarantees a specific GC partitioning scheme would provide in their context. We plan to compare the cost and benefits of different partitioning schemes: specifically, what threat models they protect against, how well they do it, what is their effect on the TCB size, difficulty of implementation and maintenance, and so forth. Further, we plan to work on a complete partitioned runtime, in addition to the GC and the allocator to provide additional services.

5 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the generous funding from the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

REFERENCES

- [1] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Jaeger Trent. 2017. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone (*In MobiSys*).
- [2] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *In ACM Computing Surveys* 51 (2019). Issue 6.
- [3] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated Partitioning of Android Applications for Trusted Execution Environments (*In ICSE*).
- [4] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to build a trusted language runtime for mobile applications (*In ASPLOS*).