Automated Detection of Cryptographic Inconsistencies in Android's Keymaster Implementations

Abdullah Imran
Purdue University
West Lafayette, Indiana, USA
imran8@purdue.edu

Antonio Bianchi
Purdue University
West Lafayette, Indiana, USA
antoniob@purdue.edu

ABSTRACT

Android smartphones use a dedicated component, Keymaster, to perform all their cryptographic, security-sensitive operations (e.g., storing cryptographic material and performing signing operations). While all Android Keymaster implementations need to expose a specific interface, their internals are hard to analyze, since their source code is generally not available. Moreover, Android Keymasters' code normally runs in a Trusted Execution Environment (TEE), where typical debugging functionality is not available. For these reasons, Keymaster implementations cannot be analyzed using white-box or gray-box automated approaches.

To address this issue, in this paper, we design, implement, and evaluate AKF (Android Keymaster Fuzzer), a device-agnostic, differential, black-box fuzzer. AKF uses a dynamic grammar to test, in parallel, multiple Keymaster implementations, comparing their behavior, looking for inconsistencies. AKF can operate on different Keymaster implementations at the same time, including Keymaster implementations running on different devices and in different TEEs (e.g., ARM TrustZone and Google's Titan-M).

We evaluated AKF by running it on 6 different Android devices, where it correctly detected 87 implementation inconsistencies that are a cause for concern in terms of both security and usability of cryptographic operations, including a previously-known encryption bug affecting the Titan-M chip (CVE-2019-9465).

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**; *Hardware security implementation.*

KEYWORDS

Android, Keymaster, TEE, ARM TrustZone, Titan-M, StrongBox

ACM Reference Format:
Abdullah Imran and Antonio Bianchi. 2024. Automated Detection of Cryptographic Inconsistencies in Android's Keymaster Implementations. In *The 22nd Annual International Conference on Mobile Systems, Applications and Services (MOBISYS '24), June 3–7, 2024, Minato-ku, Tokyo, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3643832.3661893

1 INTRODUCTION

Smartphones are becoming increasingly widely used for security sensitive operations which include making financial transactions, being used as the second factor in 2FA schemes and much more. To accomplish these security sensitive tasks, smartphones have to make use of cryptographic operations such as encryption and digital signing. In modern Android phones the execution of these security-critical cryptographic functions is handled by a Keymaster. In Android, a Keymaster is a software component (backed, in many cases, by dedicated hardware components) exposing a common interface implementing cryptographic functionality, such as generating keys, signing, and encrypting data. It also has access to raw key material and is responsible for validating access control conditions on keys. Due to the security sensitive nature of the services that Keymaster, it is of paramount importance that the implementation of Keymaster be both **secure** and **correct**.

To ensure that the Keymaster is secure even against powerful adversaries, such as those with root capabilities, the core functionality of a Keymaster usually resides inside a Trusted Execution Environment (TEE), such as ARM's TrustZone. Although safer than the code running outside TrustZone, over the years TrustZone has also been targeted by attacks [11, 19, 20, 23, 25, 27]. To counteract these attacks, modern devices are equipped with an additional TEE, running in a separate chip (e.g., Pixel Titan M [8]), normally called StrongBox. StrongBox offers a higher level of **security**, and its usage is encouraged by Android. [3]

Android Keymasters offer an extensive set of cryptographic primitives, supporting dozens of different algorithms. This complexity may potentially lead to implementation bugs that can lead to catastrophic security vulnerabilities [5]. Additionally, a Keymaster not working as intended (i.e., failing to correctly sign with a specific algorithm) will inevitably lead developers to not use it, resulting in developers using less-secure options (i.e., a TrustZone Keymaster instead of a StrongBox Keymaster). For these reasons, testing correctness of Keymaster implementations is crucial to ensure security.

Testing multiple Keymasters across multiple Android devices poses its own challenges. The code running inside these TEEs varies from vendor to vendor. Therefore, even though each Keymaster provides the same functionality, its underlying implementation can be radically different. This poses a great challenge in terms of ensuring that a Keymaster implementation is **correct** as each vendor/OEM is responsible for ensuring the correctness of their Keymaster implementation. To make matters worse, both TEEs inside one device have their own separate implementation of Keymaster. Therefore, having a unified testing mechanism for all these different implementations is crucial to ensure that the cryptographic operations provided by these Keymasters are **correct**.

Due to the complexity of the Keymaster API, off-the-shelf fuzzers are unable to fuzz effectively as they have to deal with parsing logic and inter-state dependencies. Furthermore, gray-box or white-box fuzzing is impossible due to the fact that the source code of the different Keymaster implementations is not available, and TEE based implementations cannot be debugged or instrumented on consumer devices. Additionally, existing test suites for Keymaster are not comprehensive and thorough enough to test the entire Keymaster. Finally, both test suites and off-the-shelf fuzzers fail to check for correctness of cryptographic functionality.

Prior academic work has looked into fuzzing TEEs and the TAs that reside inside them. For example, TEEzz [12] is the only fuzzer built to target TEEs on Android smartphones and their TAs but is limited to being able to target only three TEEs. It is also incapable of fuzzing StrongBox implementations. Furthermore, TEEzz does not specifically look for bugs in cryptographic implementations, but instead focuses solely on typical memory corruption bugs. On the other hand, tools like Cryptofuzz [4] that uses differential fuzzing on cryptographic implementations of multiple libraries cannot be extended to use with closed-source cryptographic implementation like Keymaster as it also functions as a gray-box fuzzer that requires code coverage information.

To address these issues, we propose AKF, a differential fuzzer for Android Keymasters that is device agnostic and can be used to fuzz any Android Keymaster regardless of device and type of TEE. AKF utilizes the Hardware Abstraction Layer (HAL) to send the same input to multiple implementations of Keymaster. To ensure efficient and effective fuzzing, it uses a dynamically-built grammar that generates type and state-aware inputs for our fuzzer. Furthermore, since Keymaster APIs often require the output of other APIs as input, our grammar dynamically infers state dependencies based between cryptographic primitives.

AKF can simultaneously fuzz multiple Keymaster implementations across multiple devices, and, by comparing the results across multiple Keymasters, AKF can reveal implementation discrepancies. A comparison with similar efforts shows that AFK is the first of its kind device agnostic fuzzer capable of fuzzing both TrustZone and StrongBox versions of Keymaster to detect incorrectness of cryptographic functionality.

To evaluate AKF and detect inconsistencies in Keymaster implementations, we run it simultaneously on multiple devices. This experiment discovered various implementation inconsistencies ranging from trivial cases, such as difference in exception handling or unimplemented algorithms, to more security critical inconsistencies, such as silent failures or random errors that result in broken cryptography i.e., a signature generated by a private key that cannot be verified by its public key. Our experiment also pinpointed a previously-known vulnerability affecting Pixel smartphones (CVE-2019-9465) [5].

We then manually analyzed each inconsistency to understand their root causes as well as their implications. We found that these inconsistencies not only have adverse effects on the security of an Android device, but also pose problems to the developers that are trying to use Keymaster functionality.

In summary, these are the main contributions of our work:

- We developed AKF [9], the first device-agnostic, automated, differential fuzzer for Android Keymasters that is able to target both TrustZone- and StrongBox-based implementations.
- We designed a dynamic grammar for the Android Keymaster HAL. Our grammar considers both, sequence of API calls, as well as the individual parameters for each API call.
- We developed an infrastructure that allows parallel and differential testing on multiple Android devices simultaneously.
- We used AKF on 10 Keymaster implementations across 6 different devices, and we found 87 implementation inconsistencies (including a previously-known vulnerability). Some of these found inconsistencies are due to incorrect or differing implementations by vendors. Others are a result of missing implementations due to vendor design choices. Some of these inconsistencies lead to lack of usability of cryptographic operations, while others are cause for security concerns.
- We detail our findings regarding the identified inconsistencies. This includes identifying reasons for the inconsistencies and their resultant effects on overall usability and security of the Android ecosystem.

2 BACKGROUND

2.1 Trusted Execution Environments

In Android, the non-secure "rich" operating system coexists with a "secure" isolated execution environment running in a Trusted Execution Environment (TEE). TEEs can be implemented in a variety of ways, such as ARM's TrustZone, which acts as the TEE for the majority of Android devices. Systems that take advantage of a TEE can store data and run code in two different contexts. The trusted kernel and trusted applications (TAs) make up the secure environment. They are both separated from the insecure environment and OEM-signed. Any code that resides inside a TEE can only be altered by the OEM. Therefore, all TA implementations, including the Keymaster TA, are done by the OEM. Furthermore, the code inside a TEE is usually kept closed source.

Android offers APIs that can interact with "trusted" applications without granting direct access to them, allowing third-party applications in the insecure world to access services that are run in the secure environment. Moreover, since Android 9, Android devices can come equipped with an additional, separate TEE. We will refer to this second form of TEE as StrongBox. The StrongBox is a separate Hardware Security Module (HSM) that has its own CPU, secure storage, random number generator, and other forms of hardware security features that make it a more secure and isolated form of TEE.

2.2 Android Keymaster

Android applications and system services that wish to make use of cryptographic and security features need to make use of them through the Keymaster. The Keymaster can be broken down into several layers. At the top level we have Android Keystore which is the main point of entry for third party applications when it comes to using Keymaster features. Android Keystore then forwards each request to the Keymaster HAL (Hardware Abstraction Layer). The Keymaster HAL is an OEM-provided, dynamically loadable library used by the Keystore service to provide hardware-backed cryptographic services. The purpose of the HAL is to ensure that regardless of the underlying hardware and OEM-provided components, the Android framework, and apps function in the same manner on every Android device by exposing a uniform and consistent API. To facilitate OEMs in their development of OEM-provided components, Android provides HAL files that contain the specification for what the OEM implementation needs to accomplish. This includes all APIs that the OEM needs to expose to the HAL and details of what the underlying implementation must achieve when an API is called.

To keep things secure, HAL implementations do not perform any sensitive operations in user space, or even in kernel space. Sensitive operations are delegated to a TEE reached through some kernel interface. In current Android smartphones this TEE is implemented using ARM TrustZone. Inside TrustZone, there exists a Keymaster Trusted Application (TA) which performs all the requisite sensitive operations. Since the Keymaster TA exists inside a TEE, it is secure from most forms of attacks including root attacks (i.e., attackers able to fully compromise the Linux kernel, running in the "non-secure" world).

The method `setIsStrongBoxBacked` is used during the generation of a cryptographic key to select whether the StrongBox Keymaster implementation should be used, instead of the TrustZone one. If this method is not called at all Android defaults to using TrustZone. Any subsequent cryptographic operation performed with a key will be carried out by the Keymaster implementation inside the TEE which stored the key.

2.3 Related Subsystems

Keymaster keys can be created in a way such that, in order to use them, user authentication is required. This means, any cryptographic operation that uses such a key, and therefore, requires the use of user authentication, needs to interact with either the Gatekeeper service or the Fingerprint Service. Both of these all have their own TAs that reside inside the TEE. The most important part of the interaction between these subsystems is the exchange of AuthTokens. AuthTokens are generated by both Gatekeeper and Fingerprint subsystems whenever a successful authentication takes place. These are then transferred to the Keystore service, which in turn forward them to the Keymaster whenever a cryptographic operation needs to take place that requires user authentication. Since many Keymaster operations can require user authentication, in order to fuzz Keymaster to the fullest extent it is essential that Gatekeeper subsystem is tested alongside them.

3 MOTIVATION & DESIGN

Security features and protocols in any computer system are almost always dependent on the use of cryptography. In Android devices, all cryptographic functionality is handled by the Keymaster. The Keymaster is responsible for handling cryptographic key material and using these keys to perform operations such as signing or encrypting data.

In order to ensure that Android security features and protocols are performing as advertised, it is essential that the underlying cryptographic functionality be tested for correctness. Incorrect or missing implementations of cryptographic functionality within a Keymaster pose a serious threat to the security of Android devices [5]. Even in cases where these incorrect implementations cannot be directly exploited by an attacker, they can still lead to scenarios where developers end up either not using certain security primitives or using substandard security practices. This can in turn lead to less secure Android apps, which is why it is essential that we test every Keymaster implementation for the correctness of their cryptographic functionality.

3.1 Challenges

In this section, we outline the primary challenges in developing a fuzzer that can effectively and efficiently fuzz Keymaster implementations on multiple devices.

C1: Testing heterogeneous implementations. Keymaster consists of multiple components ranging from the exposed HAL interface down to the trusted application (TA) that runs inside the TEE. TEE implementations, and therefore Keymaster TA implementations, are heterogeneous as they are OEM-dependent. Hence, acquiring knowledge regarding the internal working of one TEE to come up with a testing mechanism does not allow the generalization of the solution to other TEE implementations. To make matters more complicated, since Android 9, Android phones can come equipped with multiple OEM implemented TEEs (TrustZone and StrongBox). Furthermore, testing techniques, such as white-box fuzzing, cannot be applied here due to the unavailability of source code. Similarly, gray-box fuzzing, which requires some guiding principle like coverage, also cannot be employed since it is impossible to run custom, instrumented TAs in consumer devices.

C2: Generating valid test cases. Test suites designed for Keymaster, such as the vendor test suite from Android, are not comprehensive enough to thoroughly check for the correctness of its cryptographic functionality. Furthermore, the purpose of such test suites is not to check for the correctness of the underlying cryptographic functionality but rather to check if basic Keymaster functionality has been implemented. Similarly, typical off-the-shelf fuzzers also do not focus on testing for correctness of underlying cryptographic functionality but instead focus on things like memory corruption bugs. Moreover, any test input generated by a fuzzer has to successfully pass through rigorous parsing logic for the Keymaster to even consider a valid input without discarding it immediately. Furthermore, most Keymaster APIs are inter-dependent and require passing the output of one API call as the input for another API call.

C3: Cryptographic bug oracle. Another crucial challenge is to be able to test the correctness of cryptographic functionality. Most fuzzers rely on crashes to detect bugs. However, using crashes as a "bug oracle" is not effective to test cryptographic implementations, since the absence of crashes does not imply that the cryptographic functionality works as intended. For instance, a bug leading to incorrectly accepting tampered signatures does not lead to a crash, but it still has significant security consequences.

3.2 Performance of existing solutions

To the best of our knowledge no one has attempted to create a fuzzer just for Keymaster. However, there are some previous works that look into fuzzing Android TEEs.

TEEzz [12] is a TEE aware fuzzer which can fuzz Trusted Applications (TAs) inside a TEE. They achieve their fuzzing they use the OEM proprietary libraries in the "non-secure" world to infer information about the type of data that needs to be sent to the TA. Even though the TEEzz has been designed to fuzz three of the most popular TEE implementations on Android devices, it cannot cater to Keymasters on other TEEs including all StrongBox TEEs since it requires knowledge of the OEM proprietary libraries which differ for different OEMs and different TEEs. Therefore, TEEzz only partially satisfies **C1**. TEEzz does have the capability of inferring value dependencies just like our dynamic grammar so it does satisfy **C2**. Finally, TEEzz fails to satisfy **C3** as it looks only for bugs that can cause crashes. This means that TEEzz fails to identify incorrect cryptographic implementations, unless they result in a crash. For example, an incorrectly implemented signature verification algorithm could result in invalid signatures being verified successfully without triggering any crash. Unfortunately, retrofitting TEEzz to perform differential fuzzing in order to detect incorrect cryptographic implementations is not trivial, as the inputs it generates are customized for each target TEE. For this reason, using TEEzz would require changing the entire input generation process as well as adding a "comparator" component to compare the results of the tested cryptographic operations.

TEEfuzzer [14] is a coverage guided fuzzer intended for use with OP-TEE [6] only. Despite it being able to fuzz a popular TrustZone operating system in OP-TEE, it was not designed nor is it capable of fuzzing other TEE implementations or the Keymasters inside them as Keymaster implementations on real devices, unlike OP-TEE, are closed source and therefore, not suited for fuzzing with instrumentation-based coverage-guided fuzzers. Hence, TEEfuzzer is unable to achieve **C1**. Being reliant on coverage information for fuzzing, it is unable to meet the challenge of fuzzing a closed source Keymaster implementation as seen on real devices, so it fails **C2**. TEEfuzzer does not test for the correctness of the cryptographic primitives and therefore, it fails **C3**. Modifying TEEfuzzer to detect cryptographic inconsistencies is extremely challenging since we would need to convert it into a black-box fuzzer so it can be compatible with real device Keymasters. Moreover, it only detects bugs based on crashes and has no means of detecting cryptographic bugs that do not result in a crash, just like TEEzz.

The fuzzer by Melotti et al [21] for the Titan-M chip is, to the best of our knowledge, the only fuzzer for any StrongBox or HSM on an Android phone. Unfortunately, that is the only TEE it is capable of fuzzing and therefore fails to achieve **C1**. However, they do implement a structure aware black-box fuzzer that should be capable of generating inputs that can be correctly parsed and thus, it satisfies **C2**. Like every other fuzzer on this list, it also is incapable of checking for the correctness of the cryptography implemented inside the TEE and hence, fails **C3**. Since this fuzzer was designed specifically for the Titan-M chip, it cannot be readily modified to be used to target other Keymaster implementations. In addition, it still suffers from the problem of the two fuzzers discussed above in its inherent inability to detect cryptographic bugs that do not result in a crash.

Fuzzers like Cryptofuzz [4] apply the concept of differential fuzzing to test multiple cryptographic libraries for inconsistencies. Cryptofuzz requires the creation of a separate module for any new library that it needs to fuzz. Creating these modules requires knowledge of the internal working of the target library, which in the case of Keymasters is proprietary information. Therefore, Cryptofuzz only partially satisfies **C1**. Furthermore, Cryptofuzz is an instrumentation-based coverage-guided fuzzer and therefore, cannot be used with a closed source Keymaster implementation and fails to satisfy **C2**. However, Cryptofuzz is able to detect incorrect cryptographic implementations by comparing the outputs of multiple targets and therefore satisfies **C3**. For these reasons, using Cryptofuzz for fuzzing Keymasters is not straightforward. Firstly, it would require creating a module for each different Keymaster implementation. This module creation process would require knowledge of the proprietary OEM libraries for each Keymaster implementation. Secondly, real device Keymasters are closed source and therefore, cannot be fuzzed by coverage guided fuzzers like Cryptofuzz.

Table 1: Fulfillment of Challenges, ✓=Fulfilled, ✗=Not Fulfilled, ✓✗=Partially Fulfilled

Fuzzer	TrustZone	StrongBox	C1	C2	C3
TEEzz	✓	✗	✓✗	✓	✗
TEEfuzzer	✓	✗	✗	✗	✗
TM fuzzer	✗	✓	✗	✓	✗
CryptoFuzz	✗	✗	✓✗	✗	✓
AKF	✓	✓	✓	✓	✓

3.3 AKF Design

In this section, we list the design choices we made in order to overcome the mentioned challenges.

Design Choice 1: Fuzzing the HAL. As a solution to the heterogeneity of the different Keymaster implementations (**C1**), our approach fuzzes them by generating inputs at the HAL layer. This allows our fuzzer to remain device agnostic as well as have the ability to fuzz both TrustZone and StrongBox implementations. We also use the same approach to fuzz the Gatekeeper subsystem, which allows us to test Keymaster cryptographic functionality that is only used while interacting with Gatekeeper.

Design Choice 2: Dynamic Grammar Generation. In order for a fuzzer to generate test cases that not only pass through the parsing logic but are also able to explore new states (**C2**) we need a grammar. However, creating a complete grammar is time consuming and would require recreating the grammar every time some changes are enacted. Therefore, we create a dynamically-generated grammar for the HAL. The grammar generation consists of three parts. First, the grammar generator parses through the provided HAL files to infer input types for each API from the source code. Second, the grammar generator automatically creates a set of rules regarding the dependencies between each API. In order to achieve this, it compares the type of the return value of each API and checks if the same type is used as an input for some other API. In this case, our grammar generator infers that when the return type of API α matches the input type of API β that API α should always be called before API β. Lastly, during fuzzing if our heuristic determines a sequence of API calls is unable to explore a new state, the grammar generator dynamically appends a new rule in the grammar to ensure that inputs with such sequences are no longer generated.

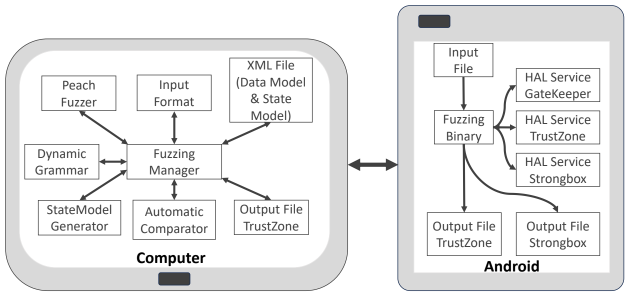


Figure 1: AKF overview

Design Choice 3: Differential testing. We detect incorrect cryptographic implementations (**C3**) of Keymaster by performing differential testing. In particular, we rely on the fact that if the same input test case was given to different Keymaster implementations, the overall result should be the same on all devices. More precisely, our fuzzer sends the same input test case to multiple different Keymasters. Note that here an input test case does not refer to a single API call but to a sequence of API calls each with their own parameters. By comparing the end results for this input test case for each Keymaster, we can determine if any of the Keymaster's deviate from the expected result. An expected result is defined as the result given by the majority of the tested Keymasters. For example, consider an input test case with a sequence of calls that starts with key pair generation, then uses the private key to sign some data, and lastly uses the corresponding public key to verify the signature. If in this case 9 out of 10 of the tested Keymasters have an end result in which everything worked correctly and the verification was successful, we would treat this as the expected result. On the contrary, the 1 Keymaster that returned a different end result (e.g., failure for the verification), will be treated as the deviant outcome. Such inconsistencies cannot be detected by typical fuzzers as they do not trigger any detectable crashes.

Table 1 summarizes how well existing solutions fulfill our design goals and requirements and how our design is able to successfully solve the aforementioned challenges.

4 IMPLEMENTATION

In this section we provide implementation details of AKF (illustrated in Figure 1).

Fuzzing Manager

Fuzzing Manager is the core component of AKF that handles all communication to and from the other components. These components run both on the target devices and the fuzzing system. We decided that the crux of the fuzzing engine needs to be implemented outside the target device for efficiency purposes. Since the fuzzing engine was on a separate system, the need arose for a manager that could handle all inter-device communication where necessary. The fuzzing manager oversees all communication to and from multiple Android devices using ADB.

We implemented the Fuzzing Manager as a Python script. To interact with the Fuzzing Binary and to send and receive files from the target Android devices it uses ADB commands. It also makes use of ADB commands to initiate Gatekeeper authentication and is responsible for automating the authentication for Gatekeeper. This authentication is necessary in order to test Keymaster functionality that is reliant on AuthTokens generated by Gatekeeper. We set the target devices to use PIN as the primary authentication method. By using ADB, we can automatically input the PIN without requiring physical interaction.

AKF uses **Peach** [7] for its input generation and mutation. Peach is a grammar-based fuzzer which generates inputs based off of a provided specification (i.e., grammar), which in our case is created by our State Model Generator.

The output of Peach is stored in an Input Format file. The input format file is essentially the fuzzing input. Due to the state dependencies between API calls, sometimes a fuzzing input requires a value that was returned by a previous API call. For example, when trying to use a key to encrypt some plaintext, the identifier for the key to be used for the encryption is part of the input. However, this identifier was returned by a previous API call to generate a key. To handle these situations, the Input Format file can sometimes have *blank spaces* that mark regions of the input that need to be filled with results of previous API calls. These *blank spaces* are filled by the fuzzing manager before it is passed on. In order to fill in the *blank spaces* in the Input Format file generated by Peach, the Fuzzing Manager retrieves the output files from the target device and extracts the requisite results necessary to fill the Input Format file before passing the Input Format file to the target device where the Fuzzing Binary uses it. Figure 2 shows an example of how this is done by the Fuzzing Manager. In this example, when Peach generates an Input Format file for the `begin()` API call, it leaves a *blank space* for the `keyBlob` as it requires the output from the `generateKey()` API call. This *blank space* is then filled in by the Fuzzing Manager before the `begin()` API is called.

It is important to note that using these *blank spaces* is crucial for efficiency as a trivial fuzzer would continue to generate random inputs to fill these *blank spaces* until it matches the value returned from the previous API call. This naive approach would result in immediate exceptions being triggered with no increase in coverage.

Key generation is the most expensive operation in terms of time, therefore, we decided to reduce that time by not creating the key every time. By reusing the same key for different sequences of calls we could save a lot of time considering that key generation on a StrongBox Keymaster took about 5 seconds. To implement this, the Fuzzing Manager keeps track of all keys created. Anytime a generated sequence of calls included a key generation call, the Fuzzing Manager first checks if that key already existed. If the key already existed, the Fuzzing Manager removes the key generation call from the Input Format file. Then the fuzzing Manager uses the previously stored key blob to fill the Input Format file if required. There is essentially only one key for each available Keymaster algorithm that existed at one time. Moreover, to ensure AKF did not completely skip over key generation testing, the Fuzzing Manager allows a key generation call to go through once every hundred times, essentially replacing the key.

Dynamic Grammar The first step of AKF's fuzzing process is the generation of the grammar. The Dynamic Grammar generator is initialized by providing it access to the HAL files of Keymaster. HAL

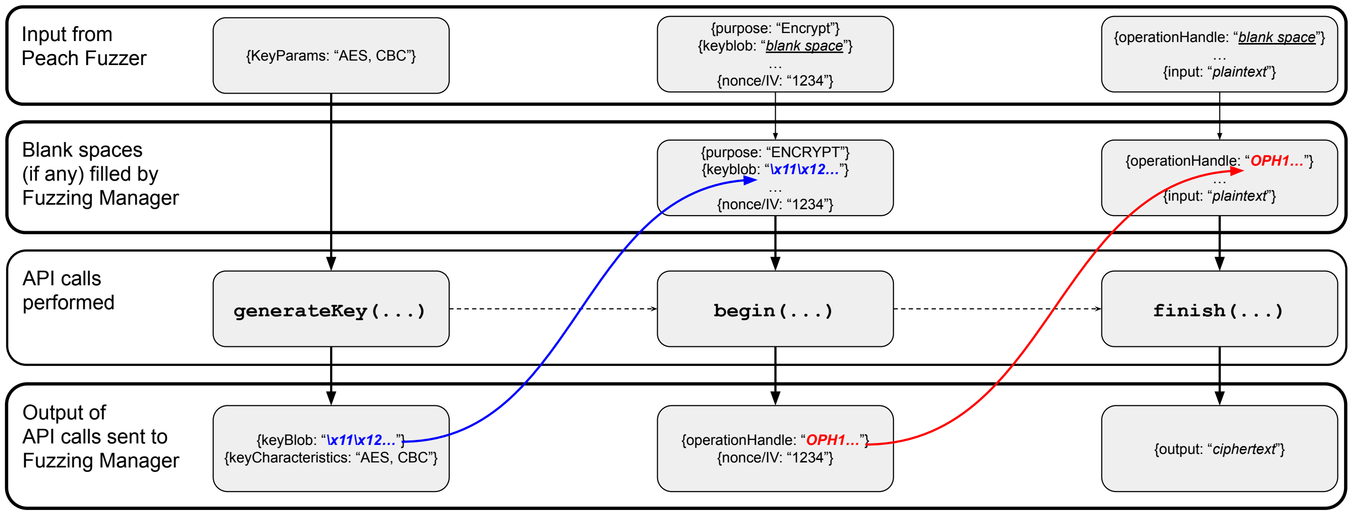

Input from Peach Fuzzer
{KeyParams: "AES, CBC"}
{purpose: "Encrypt"} {keyblob: "blank space"} … {nonce/IV: "1234"}
{operationHandle: "blank space"} … {input: "plaintext"}
Blank spaces (if any) filled by Fuzzing Manager
{purpose: "ENCRYPT"} {keyblob: "\x11\x12…"} … {nonce/IV: "1234"}
{operationHandle: "OPH1…"} … {input: "plaintext"}
API calls performed
generateKey(...)
begin(...)
finish(...)
Output of API calls sent to Fuzzing Manager
{keyBlob: "\x11\x12…"} {keyCharacteristics: "AES, CBC"}
{operationHandle: "OPH1…"} {nonce/IV: "1234"}
{output: "ciphertext"}

Figure 2: An example of how inter-state dependencies are resolved by the Fuzzing Manager. The input produced by Peach has *blank spaces* that mark regions that require output values of previous API calls that are filled by the Fuzzing Manager.

files include all the necessary details required to create a working implementation for that HAL. The most important details for us here are the API calls, their input parameters and types, and lastly their return type. By using this information, the Dynamic Grammar generator comes up with two sets of initial rules.

The first set of rules pertain to each individual API call. These rules define the type of input that is expected by each API call. For this first set of rules, the Dynamic Grammar generator converts the HAL file into a Peach Pit file, which contains the grammar Peach uses to generate input. This conversion is done by parsing the parameters of each API call within the HAL file and converting them into DataModel elements within the Peach Pit file. This procedure allows us to program Peach to generate inputs that match the parameter types required by the Keymaster API. In some cases, the parameter type for an API defined in the HAL file can be a custom data type. In these cases, the Dynamic Grammar generator searches for that custom data type within the "types.hal" file (one of the Keymaster HAL files), which contains the definition of the custom data types. By checking the definition of the custom data type, we can convert the custom data type into default data types and then into DataModel elements within the Peach Pit file.

The second set of rules pertains to the sequence of calls that will be used in one fuzz input. To create this initial set of rules for the sequence of calls, the Dynamic Grammar generator checks if the input type for an API call is the return type of some other API call within the HAL file. This allows our Dynamic Grammar to infer basic state dependencies between API calls such as *"a key must be generated before encryption"*.

After the initial set of rules have been established by the Dynamic Grammar, AKF continuously adds more rules depending on the results of the fuzzing. The Dynamic Grammar uses a heuristic to determine which new rules to add. To achieve this, it receives the input test cases and their corresponding results from the **Fuzzing Manager**. The heuristic is then used to ascertain if the results are significantly different by doing a bitwise comparison of the result of each API call as well as the overall result. If the results from each Keymaster are significantly different from each other, then, the Dynamic Grammar determines that the input fuzzed a new state and therefore inputs like this one are more useful. Sequence of calls that are unable to generate any new result even after being used as an input multiple times are considered stale. The Dynamic Grammar stores a list of the last hundred sequences of calls that are stale and continuously uses this list to update a frequency table. This frequency table counts how many times a subsequence of two API calls occurs within the 100 stored stale sequences. Once the frequency goes past a certain threshold the Dynamic Grammar considers that particular pair of API calls as the likely culprit for causing staleness, Therefore, it establishes a new rule disallowing a sequence of calls that includes a subsequence of that pair of API calls.

For AKF, the Dynamic Grammar generator is used both for Keymaster as well as Gatekeeper so that a grammar exists for both HAL interfaces. This is necessary in order to allow AKF to fuzz Keymaster functionality that requires interactions with Gatekeeper.

Fuzzing Binary The fuzzing binary is responsible for handling all actions inside the target device itself. Its main responsibility is to send the same input to the multiple Keymaster, record their responses, and return them. We designed it to be a system service so that it can interact with the Keymaster HALs directly. We chose to fuzz the Keymaster through the HAL layer as it is the deepest layer in the Android ecosystem before the implementation of Keymaster becomes OEM-dependent. Furthermore, the fuzzing binary is also responsible for interacting with the Gatekeeper HAL and sending any AuthTokens generated by Gatekeeper to each Keymaster HAL (Section 2.3).

The fuzzing binary is implemented as an Android system service capable of interacting with the HAL layer. It is a simplistic service that reads the input file sent by the Fuzzing Manager and uses that as the input to send to the HAL layer. It is able to interact with both

Keymaster HAL as well as Gatekeeper HAL. In order to deploy it to a target Android device we need to root the device.

State Model Generator This is the first step in creating a fuzz input within AKF. Since AKF is a differential fuzzer, each fuzz input needs to test not just one single API call within Keymaster, but instead it needs to test a sequence of calls to see if it generates the same output at the end of the entire sequence of calls. Therefore, each fuzz input is a sequence of calls that is going to be tested in a single run.

A problem the State Model Generator has to deal with is the infinite length for a sequence of calls. To counter this problem, we decided to set a limit for the number of single API calls that are allowed within a sequence of calls. In order to determine this limit, we first observed how Keymaster APIs were called when cryptographic operations were used by an Android app. Our observation showed that for any complete cryptographic operation (i.e., a verification after signing or decryption after encryption) only used at most two calls of a single API. Based on this observation our State Model Generator limits the number of single API calls that can be inside a sequence of calls to three, i.e., a call to `generateKey()` can only be made three times within a sequence of calls. This helps us limit the total number of fuzz inputs that can be produced and still effectively test edge cases where multiple uses of a single API call might trigger a bug.

We also implemented the State Model Generator as a Python script. It uses randomization to select the API calls to add to a sequence of calls. It counts the number of times it has added one API call to the sequence to ensure it does not go over our defined limit of three. The generated sequence of calls is then compared to the current set of rules established by the Dynamic Grammar. If the generated sequence of calls does not violate the rules it is then used to populate the XML File that is used by the Peach Fuzzer to generate the Input Format file. If it is in violation of the grammar rules, the generated sequence of calls is discarded, and a new generation process is started from scratch. Furthermore, the sequence of calls generated by the State Model Generator is also able to cater to Gatekeeper API calls. This allows AKF to fuzz both Keymaster and Gatekeeper together with a single sequence of calls that contains both Gatekeeper and Keymaster API calls.

Automatic Comparator The automatic comparator is the last step in the AKF fuzzing process. The purpose of the comparator is to compare and differentiate between responses returned by multiple Keymasters (**G2**). It is the essential step in determining whether there is some implementation inconsistency that exists between two implementations of the same feature.

Some Keymaster APIs return values in a non-deterministic way. For example, an encryption operation on the same plaintext will return different ciphertexts on each Keymaster. The happens because the ciphertext is dependent on the raw value of the key being used which is again different for each Keymaster. A difference in the returned ciphertext in such cases is not an inconsistency. Therefore, naively comparing return values will result in false positives.

To overcome this challenge, the return value of each API call is subject to three distinct levels of comparisons. Firstly, all API calls are expected to have the same return type regardless of the Keymaster being used. This comparison detects inconsistencies that result in exceptions or some other error in execution.

Table 2: List of used Android Devices with their available Keymasters (KMs) and Android Versions, FN=Factory New, U=Updated

ID	Device Name	KMs	Version
D1	Pixel 3 (FN)	TZ & SB	Android 9
D2	Pixel 3a (U)	TZ & SB	Android 12
D3	Pixel 5 (U)	TZ & SB	Android 13
D4	Samsung S21 (FN)	TZ & SB	Android 12
D5	Xiaomi 11 Lite (FN)	TZ	Android 11
D6	Hikey960	TZ	Android 9

The second level of comparison is comparison of return value format. This level of comparison is for API calls that are expected to have return values with different raw values but matching format on each Keymaster. For example, an encryption operation will return different ciphertexts but the size of the ciphertext should match.

The third level of comparison is raw return value comparison. API calls that are expected to return matching values are subject to this level of comparison. This includes operations like decryption which should return the same plaintext for each Keymaster.

The automatic comparator is programmed with a set of rules regarding what level of comparison to use for each API call. This allows the comparator to determine inconsistencies in expected behavior while keeping false positives at a minimum.

5 EVALUATION

We designed our evaluation to answer the following questions:

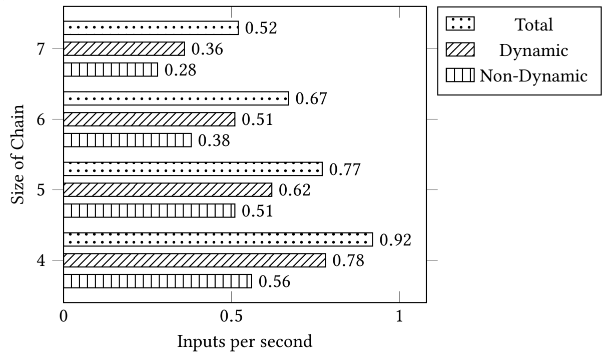
- **RQ1:** How efficient is AKF at generating valid test cases using the dynamic grammar?
- **RQ2:** Can AKF find implementation inconsistencies?
- **RQ3:** How long does it take for AKF to expose implementation inconsistencies?
- **RQ4:** How much code coverage can AKF achieve?

For **RQ1**, we study the throughput of valid cases from AKF. Next, for **RQ2**, we ran AKF on 6 different Android devices to look for implementation inconsistencies in their Keymasters. To answer **RQ3**, we ran AKF 5 times on the same device to determine how long it takes on average to find one known inconsistency. Finally, for **RQ4**, we ran AKF on OP-TEE, an open-source TEE implementation, to evaluate the coverage achieved by AKF. Experiments for **RQ1** and **RQ3**were conducted with a Google Pixel 3 while the fuzzing machine had an Intel i5-11400F processor with 16 GB Ram. The experiment for **RQ2** made use of multiple devices as listed in Table 2. The experiment for **RQ4** was run on a HiKey960 board with 4 GB Ram.

5.1 Performance

To evaluate the performance of AKF (**RQ1**), we measure its throughput first. We measure both the total number of inputs being generated as well as the number of valid inputs being generated. We classify an input as valid if it results in an exception-less execution for at least one of the Keymasters under test. We partition the data between varied sizes for the sequence of calls. This is necessary

Figure 3: Throughput (measured as inputs per second) of valid inputs generated by AKF with and without the use of Dynamic Grammar.



because, as the sequence of calls gets longer, the throughput decreases. This is expected as each additional API call requires its own additional processing time. To evaluate the performance of the dynamic grammar we then do an ablation study in which we disable the dynamic grammar generation. To achieve the partitioning between varied sizes for the sequence of calls we force our State Model generator to only generate sequence of calls with a fixed size. We ran each size of chain for 10 minutes for both Dynamic Grammar inputs and Non-Dynamic Grammar inputs and calculated the rate of inputs being processed per second. For the total number of inputs, we used the combined rate of both Dynamic Grammar inputs and Non-Dynamic Grammar inputs that were sent over 20 minutes total. It should be noted that there was virtually no difference between the rate of total inputs processed when running the experiment with or without Dynamic Grammar as it has no effect on the total inputs processed but rather on the percentage of valid inputs.

Figure 3 summarizes our results. As the table shows, the proportion of valid test cases increases significantly when using dynamic grammar generation. For instance, for a sequence of calls of size 4, the proportion of valid inputs to total inputs was 0.85 with the dynamic grammar, whereas, without the dynamic grammar, the proportion drops to 0.63. Overall, our dynamic grammar approach increases the performance of AKF by 26% percent.

Answer to RQ1: AKF's throughput of valid test cases increases by 26% due to the use of dynamic grammar.

5.2 Discovery Results

Then, we evaluated AKF by running it on multiple devices to check for implementation inconsistencies between their Keymasters (**RQ2**). We chose a variety of devices covering different vendors, types of Keymaster implementations (TrustZone vs. StrongBox), and Android versions, resulting in a total of 6 devices and 10 Keymaster implementations (Table 2).

Some devices we labeled as Factory New (FN). These devices are those that were never updated out of the box. Therefore, these devices contain Keymaster implementations that have not been updated. This allows us to discover implementation inconsistencies that existed in previous versions of some devices that may not exist in an updated device (U).

Table 3: Summary of the types of inconsistencies found. Note that some inconsistencies can be categorized into multiple types.

Inconsistency Type	No. Found
Inconsistency among different Keymasters (⧫)	68
Inconsistency within the same Keymaster (★)	24
Inconsistency with respect to expected behavior (■)	9
Total Unique Inconsistencies	87
due solely to missing implementations	*34*

Table 4: Basic Block Coverage achieved by the Vendor Test Suite, AFL, and AKF

Test/Fuzzer	% BB Coverage Achieved After 24 Hrs	
	Keymaster	Gatekeeper
Vendor Test Suite	26.4	33.1
AFL	8.1	11.6
AKF	37.7	49.3

We divide the implementation inconsistencies we found into three categories. The first category is inconsistencies among different Keymasters (⧫). These are inconsistencies that exist due to differences in OEM implementations such as different exception messages across devices for the same problem. For example, device **D4** triggers an exception at the time of key generation when `SHA512withECDSA` is used. Whereas **D1**, **D2**, and **D3** allow the generation of the key, but trigger an exception when the key is used for signing.

The second category of inconsistencies is inconsistencies within a single Keymaster (★). These are inconsistencies that exist due to a difference of behavior within the same implementation for similar test cases. For example, when handling unsupported algorithms, StrongBox Keymaster on device **D4** does not allow the generation of the key. However, only in the case where the algorithm is set to `SHA1withRSA`, it allows the key to be generated but throws an exception when someone attempts to use the key. We believe these inconsistencies are most likely due to negligence of the OEM, which results in one scenario being handled completely differently than all other similar scenarios.

The last category of inconsistencies is inconsistencies with respect to the expected behavior (■). We define expected behavior as the normal execution of a cryptographic operation. For example, it is expected behavior that any ciphertext generated by a public key should be decipherable by the corresponding private key. An example of this inconsistency would be if the generated ciphertext could not by decrypted by the private key used when generating it. These are the more dangerous inconsistencies as the OEM Keymaster implementation fails to perform a cryptographic operation in the correct manner, which may lead to an exploitable vulnerability. Moreover, it can cause cryptographic schemes using specific algorithms to operate incorrectly in some devices.

Table 3 summarizes the number of inconsistencies we found, while Table 5 details the full list of implementation inconsistencies we found in all the analyzed Keymaster implementations.

Table 5: Implementation inconsistencies found by AKF in the 10 tested implementations. TZ=TrustZone, SB=StrongBox, DX=Device, ♦=Inconsistency among different Keymasters, ★=Inconsistency within the same Keymaster, ■=Inconsistency with respect to Expected Behavior. Circled numbers indicate the corresponding findings (as detailed in Section 6).

	D1		D2		D3		D4		D5	D6
	TZ	SB	TZ	SB	TZ	SB	TZ	SB	TZ	TZ
ECDSA Signature Algorithms										
NoneWithECDSA										
SHA1WithECDSA								❹♦★		
SHA224withECDSA		❺♦		❽♦		❽♦		❸♦		
SHA256withECDSA										
SHA384withECDSA		❽♦		❽♦		❽♦		❸♦		
SHA512withECDSA		❽♦		❽♦		❽♦		❸♦		
RSA Signature Algorithms										
MD5withRSA		❺♦		❽♦		❽♦		❸♦		
NONEwithRSA		❶■						❶■		
SHA1withRSA								❹♦★		
SHA1withRSAPSS		❺♦		❶■		❶■		❹♦★		
SHA224withRSA		❺♦		❽♦		❽♦		❸♦		
SHA224withRSAPSS		❺♦		❽♦		❽♦		❸♦		
SHA256withRSA										
SHA256withRSAPSS										
SHA384withRSA		❽♦		❽♦		❽♦		❸♦		
SHA384withRSAPSS		❽♦		❽♦		❽♦		❸♦		
SHA512withRSA		❽♦		❽♦		❽♦		❸♦		
SHA512withRSAPSS		❽♦		❽♦		❽♦		❸♦		
RSA Encryption Algorithms										
RSAECBNoPadding										
RSAECBOAEPPadding										
RSAECBOAEP/SHA1/MGF1	❻★	❻★	❻★	❻★	❻★	❻★	❻★	❻★	❻★	❻★
RSAECBOAEP/SHA224/MGF1		❺♦❼■		❽♦		❽♦		❸♦		
RSAECBOAEP/SHA256/MGF1		❼■		❼■		❼■				
RSAECBOAEP/SHA384/MGF1		❽♦		❽♦		❽♦		❸♦		
RSAECBOAEP/SHA512/MGF1		❽♦		❽♦		❽♦		❸♦		
RSAECBPKCS1Padding										
AES Encryption Algorithms										
AESCBCNoPadding										
AESCBCPKCS7Padding										
AESCTRNoPadding										
AESECBNoPadding										
AESECBPKCS7Padding										
AESGCMNoPadding		❷■								
HMAC Algorithms										
HmacSHA1										
HmacSHA224		❺♦		❹♦★		❹♦★		❹♦★		
HmacSHA256										
HmacSHA384		❹♦★		❹♦★		❹♦★		❹♦★		
HmacSHA512		❹♦★		❹♦★		❹♦★		❹♦★		

Answer to RQ2: AKF found 87 unique implementation inconsistencies.

We note that the detected inconsistencies are due to multiple reasons, which will be discussed more in detail in Section 7.1. One of these reasons is missing implementations. We can see from Table 3 that 34 out of the 87 detected inconsistencies are classified as such solely due to missing implementations. As discussed ahead in Section 6 finding ❽, these detected inconsistencies can be considered as "false positives", leaving the other 53 detected inconsistencies as "true positives".

5.3 Discovery Efficiency

To evaluate AKF efficiency (**RQ3**), we run it till it finds a known implementation inconsistency and measure the time taken. We do this 5 times and average the time taken to account for randomness. The inconsistency we chose (which we will explain in detail in Section 6) was signature verification using `SHA1withRSAPSS` in a Pixel 3 StrongBox Keymaster.

AKF is able to consistently find the inconsistency with an average time of 23 hours and 46 minutes with a standard deviation of 2 hours and 7 minutes. In comparison AKF without the use of dynamic grammar finds the inconsistency in an average of 26 hours and 11 minutes with a standard deviation of 2 hours and 53 minutes.

Answer to RQ3: AKF can find potentially dangerous implementation inconsistencies within a day.

5.4 Code Coverage

To measure code coverage, we focus on the only TEE implementation for which source code is available, i.e., OP-TEE. For the purposes of measuring coverage, we used a modified OP-TEE version that uses AFL-style instrumentation to measure achieved code coverage within TAs, by populating a coverage bitmap [1].

We first ran AKF on the modified OP-TEE for 24 hours. We also ran Android's vendor test suite for Keymaster and Gatekeeper and measured its coverage to use as a baseline for comparison. We then also ran AFL [2] for 24 hours to have a comparison against an off-the-shelf fuzzer. The results of our coverage measurement are shown in Table 4. The table shows that AKF achieves a higher coverage than the baseline vendor test suites and existing fuzzers, like AFL.

Answer to RQ4: AKF provides significantly more code coverage than any existing fuzzer or test suite.

6 FINDINGS

In this section we detail the inconsistencies AKF found, listed in Table 5.

❶ **Failure to verify signature.** We identified two signature algorithms, `NONEwithRSA` and `SHA1withRSAPSS`, where the signature that was generated by a Keymaster could not be verified by the same Keymaster. The problem with `NONEwithRSA` existed on all 3 Pixel Devices as well as the Samsung Device (**D1, D2, D3, D4**). We reported this to both Google and Samsung. Google later fixed this issue, and the algorithm now works as intended on the newer and updated Pixel devices.

The problem with `SHA1withRSAPSS` exists on the StrongBox Keymaster of the newer and updated Pixel devices (**D2, D3**). Interestingly, this problem does not exist in the factory new Pixel 3 (**D1**) which suggests that this is a regression bug, caused by an update. We reported this bug to Google who classified it as infeasible, and it still remains to be fixed. However, as we will see below (❺), classifying it as infeasible is incorrect.

These bugs are examples of inconsistencies with respect to the expected behavior (■). Since the algorithm is able to create a signature in the first place, the developer clearly intended for the algorithm to work for verification as well. The verification, however, fails without giving any exception. Upon further analysis we identified the signature being generated was correct and the problem was with the verification. Due to the silent nature of the failure, i.e., there is no exception or warning for a developer to indicate that the algorithm might not work with these Keymasters, any Android app that uses these algorithms might encounter unexpected problems that may cause some denial of service on apps running on these devices. Furthermore, it might be possible that the verification implementation ends up incorrectly verifying a signature allowing for an adversary to take advantage.

❷ **Keystore operation failed on decryption.** AKF was able to reproduce the bug in CVE-2019-9465 [5]. Even though the bug has been patched on the newer and updated models of Google Pixel phones, the bug still exists on the unupdated Pixel 3 (**D1**). The problem lies in the implementation of the `AESGCMNoPadding` algorithm in the StrongBox Keymaster which causes the ciphertext that is produced to be incorrect, resulting in the inability to decipher it.

❸ **Ill-timed exceptions.** We found that if an app tries to use the StrongBox Keymaster on Pixel devices (**D1, D2, D3**) using an algorithm that is not supported, the Keymaster still allows the app to create the key. Only afterward, when the developer would try to use the key, the app would get an "Invalid Argument" exception. In contrast, the Samsung (**D4**) StrongBox Keymaster implementation does not allow the key to be generated at all (except for a couple of cases that we will discuss next) and gives an exception when an app tries to generate a key. This is an example of an inconsistency between Keymasters (♦) as the StrongBox Keymaster of two different OEMs handle the same situation differently. It must be noted though, that the Samsung (**D4**) way to handle unsupported algorithm makes more sense as the Pixel (**D1, D2, D3**) StrongBox Keymaster allows the developer to create a key despite the fact that an app would be unable to use it, resulting in a waste of resources and confusion for an app developer.

❹ **Inconsistent exceptions.** We found that in both the Samsung (**D4**) StrongBox Keymaster and the Pixel (**D1, D2, D3**) StrongBox Keymaster, there were cases where the exception being generated was not consistent with the exceptions seen elsewhere within the same Keymaster (★).

For example, in Samsung (**D4**) StrongBox Keymaster, the exception to indicate the algorithm was unsupported normally came at the time of key generation when an unsupported algorithm was used. However, in the case of signature algorithms using SHA-1 digest, this exception came at the time of signing. Furthermore, in the case of `SHA1withECDSA` signing algorithm, the exception message was completely different to the rest. Normally, an "Unsupported Digest" exception would be triggered, at the time of key generation when an unsupported algorithm was used. However, for this particular algorithm, it is an "Invalid KeyBlob" exception at the time when signing was attempted. Moreover, when the unsupported algorithm was an HMAC algorithm, the exception triggered at the time of key generation would be unlabeled (i.e., "-59") in the case of the Samsung (**D4**) StrongBox Keymaster.

Pixel (**D1, D2, D3**) StrongBox Keymaster usually gives an exception when the developer tries to use a key for an unsupported algorithm, instead of earlier when the developer generates the key. However, in the case of unsupported HMAC algorithms, the exception is thrown immediately, at the time of key generation.

❺ **A regression bug.** We found that the StrongBox Keymaster in the factory new Pixel 3 (**D1**) actually had 2 advantages over its newer/updated family devices (**D2, D3**) (♦). Firstly, it supported all algorithms with SHA-224 and MD5 as the hashing algorithm. This was apparently removed later on for unknown reasons. Secondly, it still had a working implementation for `SHA1withRSAPSS`. The same algorithm cannot be used in the more modern devices (**D2, D3**) where it is unable to verify signatures as described above (❶). This is interesting as the bug report we had submitted for this particular algorithm to Google was classified as "infeasible", even though there is a working implementation of this algorithm available in a previous version (**D1**).

❻ **A complex algorithm.** We found that the encryption algorithms `RSAECBOAEP/SHA/MGF1padding` have inconsistent implementations in most Keymasters. The algorithm would function correctly when used with SHA-1, however, using the SHA-2 family of hashes resulted in an exception at the time of decryption (★). Only the Pixel (**D1, D2, D3**) StrongBox Keymaster was able to implement this algorithm correctly when the SHA-2 family of hashes were used (❼). Even the TrustZone Keymaster implementations on all 6 would fail with an exception. We reported this bug to both Google and Samsung. Google did not provide any worthwhile response, however, Samsung clarified that these algorithms could be used with SHA-2 family of hashes if special parameters were specified when using the algorithm. We confirmed that this was indeed correct, however, we still consider this an inconsistency within the same Keymaster (★) as the SHA-1 variant of the algorithm does not require these special parameters.

❼ **Working Unexpectedly.** As mentioned, we found that using the encryption algorithms `RSAECBOAEP/SHA/MGF1padding` with the SHA-2 family of hashes without specifying additional parameter specifications usually resulted in an exception. However, the Pixel (**D1, D2, D3**) StrongBox Keymaster was able to implement this algorithm correctly when the SHA-2 family of hashes were used (■). This case is interesting for two reasons. Firstly, this is the only case where the inconsistent Keymaster probably has the correct implementation. Secondly, even though we reported this inconsistency to Google they still did not feel it was important to make the implementation consistent between their own TrustZone and StrongBox Keymasters. This results in a developer being able to correctly use this algorithm in a Pixel StrongBox but not in a Pixel TrustZone.

❽ **Inconsistencies due to missing implementation.** We found that these inconsistencies were detected solely due to the missing implementation of the algorithm within the StrongBox Keymaster. We consider these inconsistencies as false positives of our fuzzer, since we speculate that developers may have intentionally decided not to implement such algorithms.

Nevertheless, we still find these cases interesting since they show that there are cases in which OEMs have decided not to support some algorithms in one device on one Keymaster but not the other. In addition, we note that the exceptions triggered when a developer attempts to use these unsupported algorithms are inconsistent and hard to interpret. For instance, the Pixel Strongbox gives an "Invalid Argument" exception if the algorithm is not supported which causes confusion for the developer as it does not inform the developer about the missing implementation and lack of support for this particular algorithm. As we will discuss in Section 7.3, we believe that developers should have clearer ways to understand when an algorithm is supported or not by a specific Keymaster.

7 DISCUSSION

In this section we discuss possible reasons for the found inconsistencies and possible ways to make cryptographic operations in Android devices more secure and usable.

7.1 Potential reasons for inconsistencies

The major reason for inconsistencies we found was missing implementations of algorithms. The reason behind having limited number of algorithms supported might be due to the limited space available for code in StrongBox. However, the results in Section 5.2 suggest that this might not always be the case. For example, the Factory New Pixel 3 was able to support algorithms that used SHA-224 as the digest, whereas the newer/updated Pixel phones no longer support SHA-224 algorithms in their StrongBox Keymaster implementations. All Pixel phones use the same Titan-M chip; therefore, it appears that at least one more digest and its related algorithms could be supported. Furthermore, the design for the StrongBox implementation in Pixel phones is extremely limited as they chose to support only two hashing algorithms, namely SHA-1 and SHA-256. Ironically, one of these algorithms, namely SHA-1, has been proven to be broken by Google researchers themselves [26]. It would be our recommendation that SHA-1 algorithms be declared obsolete not only for StrongBox Keymasters but from all Android devices.

7.2 Limited usage of StrongBox Keymasters

One key factor in making cryptographic operations in Android more secure is to use the most secure available Keymaster, which is the StrongBox Keymaster. We conducted a small-scale analysis of 1,048 popular Android on Google Play Store within the Business, Finance, Tools, Communications, and Medical categories to determine how often the StrongBox Keymaster is used by apps. We found that despite being available since Android 9, most apps are still not using the StrongBox Keymaster. This can be due to multiple reasons.

Firstly, as we saw in Section 5.2, the number of algorithms available for use with the StrongBox Keymaster implementation are very limited in comparison to the TrustZone Keymaster. Therefore, any developer looking to use an algorithm not supported by the StrongBox Keymaster would be forced to use the TrustZone Keymaster. Another possible reason for the limited usage of StrongBox could be that average Android developers have limited knowledge of security practices and cryptography [18]. Therefore, if a developer was asked to use a cryptographic operation, they would try to adopt what is most-widely used regardless of the availability of more secure options. For example, for signing, one of the most secure options would be to use `SHA512withRSAPSS` with a StrongBox Keymaster. However, due to this algorithm not being available for use with the StrongBox Keymaster, the developer now has to make an almost impossible decision. They can choose to go for the more secure algorithm in `SHA512withRSAPSS` but use it with a TrustZone Keymaster, or they could use a StrongBox Keymaster and use a less secure algorithm. Even security experts might have trouble deciding which option is more secure. Therefore, Android developers with limited knowledge of cryptography may decide to just avoid using StrongBox.

One more plausible reason developers may not use StrongBox Keymaster is that they might have encountered one of the many inconsistencies in the StrongBox Keymaster implementations we saw in Section 5.2. A developer using an algorithm that worked completely fine with the TrustZone Keymaster could encounter issues (ranging from exceptions to incorrect cryptographic operations) when attempting to use the same algorithm with a StrongBox Keymaster. This would prevent the developer from switching to using a more secure Keymaster unless the developer decides to change the algorithm they use, which would require changes not only on the app but also on any back-end server that is reliant on that cryptographic algorithm. Therefore, it is probable that in such scenarios the developers decide it is more beneficial to stay with the TrustZone Keymaster.

7.3 Recommendations

Motivated by the results, and by the low usage of StrongBox, we make the following recommendations to improve the security and usability of Keymaster implementations.

As we saw in Section 5.2, when it came to different OEMs, their Keymaster implementations would often behave differently from each other. This is problematic from a developer's point of view who does not understand why the same code runs differently on different Android devices. Furthermore, there is no way for a developer to know which algorithms are supported by a Keymaster on a certain device. In fact, this information is not available in the official Android documentation, and the inconsistent exceptions (❹) provided by different Keymaster implementations can further confuse developers. Hence, we recommend Android to lay out guidelines as to how these situations should be handled. For example, Android could specify as a mandatory requirement for a Keymaster implementation that in the case of an unsupported algorithm the exception must be consistently called "Unsupported algorithm" and should be thrown as soon as the developer attempts to generate a key to be used with that algorithm.

Additionally, we can make things easier for developers by automatically using the most secure option. Allowing developers to choose the algorithm they wish to use might be useful for compatibility purposes, however, this is not true when it comes to choosing Keymaster. The Android framework should automatically choose the most secure available TEE for the chosen algorithm. This would ensure that the most secure form of Keymaster is used regardless of the algorithm chosen. On the contrary, the default Keymaster (i.e., if the developer does not specify a Keymaster) is the TrustZone Keymaster.

Lastly, we saw that despite these implementations being deployed in the real world for a long time the inconsistencies we detected were never found before. Despite all the test suites employed by the OEMs/vendors, incorrect implementations have continued to go unnoticed. This shows that there is a clear need for better testing when it comes to Keymaster implementations, potentially using automated approaches, such as the one we implemented in AKF.

8 LIMITATIONS

One way in which AKF is limited in fuzzing Keymaster is that it is unable to fuzz the interactions of the Fingerprint subsystem with Keymaster. Although in theory it should be no different from the interactions with the Gatekeeper subsystem as almost all interaction between the subsystems is based on the use and exchange of AuthTokens generated by both Fingerprint and Gatekeeper. Nevertheless, there could still be some implementation inconsistencies within the Fingerprint subsystem that may lead to some errors in Keymaster when they interact.

Therefore, one aspect of our future work is to be able to automate biometric authentications on Android phones therefore allowing us to do automated testing of the Fingerprint subsystem and other related subsystems that interact with Fingerprint subsystem like Keymaster. Since the biometric authentication directly takes place inside a TEE, it is hard to come up with a solution that might be able to automate this problem from a software perspective without reverse engineering the TEE software. But this solution cannot be utilized across all devices as it relies on TEE knowledge which is different for each device. To achieve automated biometric authentication, we can employ hardware-based techniques where a robot could be designed to provide biometric authentication when prompted.

Another limitation of AKF is that there is no limit to the size of the sequence of calls. For example, generating a key ten times in a row in considered a valid sequence of calls. To deal with this issue the State Model Generator limits the use of a single API call. Therefore, it is possible that AKF was unable to detect some implementation inconsistencies that could only be detected when single API was called more than three times in a single sequence of calls.

Furthermore, the AKF requires that a device be rooted. This not only inhibited our ability to conduct experiments on more devices but also means that AKF can only work on off-the-shelf devices that can be rooted. In order for AKF to work on more devices it requires assistance from OEMs. Since OEMs can always root their own device, it is our recommendation that AKF be utilized by OEMs when making changes to their Keymaster implementation as an efficient method to determine whether or not their implementation is consistent with other Keymaster implementations including the ones on the same device.

9 RELATED WORK

Fuzzing has been widely used to find vulnerabilities in several types of software, including operating systems, web browsers, and network protocols. However, the use of fuzzing in the context of TEEs is still an emerging area of research.

The TEEzz [12] framework is designed to perform fuzz testing on TAs running on Android devices using dynamic instrumentation to generate and execute inputs. TEEfuzzer [14] uses heuristic seed generation to fuzz OP-TEE. Huang et al. [16] built a system for fuzzing OP-TEE which uses their own purpose-built seed generation techniques as well as automatic bug reproduction. Melotti et al. [21] are the first to look into a StrongBox TEE by doing a security analysis of Google Titan-M chip which serves as the StrongBox TEE for all Pixel phones. Shakevsky et al [24] look into Samsungs' TrustZone TEE and Keymaster and found some serious vulnerabilities in the implementation of Keymaster. Partemu [15] is a TEE OS emulation technique that can be used to fuzz TAs that reside inside TEEs without actually requiring a hardware device. Unfortunately, this technique only works when TAs are unencrypted, which is usually not the case. Another common technique to fuzz TEEs is to use a device driver interface as all TEEs must eventually communicate with the "Normal World" using driver interfaces. Difuze [13] is an interface aware fuzzer that can fuzz kernel drivers which can be extended to fuzz TEEs using the TEEs' kernel driver interface. Overall, these works demonstrate the importance of fuzzing in the context of TEEs and the potential for finding previously unknown vulnerabilities in these systems.

Differential fuzzing has been used to test for bugs by comparing results instead of requiring explicit crashes. JIT-PICKER [10] is a tool which does differential fuzzing for JavaScript engines by using state probes to compare the behavior of the JavaScript interpreter and the JIT compiler. DifuzzRTL [17] is a differential fuzzing tool that detects CPU RTL vulnerabilities by comparing the execution of an RTL design against a golden model. NEZHA [22] is an efficient input format agnostic differential testing framework which tracks relative behavioral differences between multiple programs. Cryptofuzz [4] is a tool designed to do differential fuzzing on multiple cryptographic libraries to look for implementation inconsistencies These works have shown that differential fuzzing is an effective tool to find bugs especially when the purpose it to look for correctness of implementation.

10 CONCLUSION

This paper presents AKF, a device-agnostic fuzzer capable of effectively fuzzing both TrustZone and StrongBox variants of Android Keymasters, looking for improper implementations of cryptographic primitives. Using AKF, we found 87 unique inconsistencies that we manually analyzed further to discover their implications in terms of usability and security. Our analysis showcases the limitations of previous testing mechanisms in regard to detecting cryptographic inconsistencies within Keymaster implementations. Additionally, the found inconsistencies not only lead to unexpected behaviors or potential vulnerabilities but can also force developers to use less secure cryptographic primitives.

ACKNOWLEDGEMENTS

This material is based upon work supported in part by Google's ASPIRE Award. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Google.

REFERENCES

[1] Fuzzing OP-TEE with AFL. https://static.linaro.org/connect/san19/presentations/san19-225.pdf, 2019.

[2] AFL. https://github.com/google/AFL, 2023.

[3] Android Security Best Practices. https://source.android.com/docs/security/best-practices/hardware, 2023.

[4] Cryptofuzz. https://github.com/guidovranken/cryptofuzz, 2023.

[5] CVE-2019-9465. https://alexbakker.me/post/mysterious-google-titan-m-bug-cve-2019-9465.html, 2023.

[6] OP-TEE. https://www.trustedfirmware.org/projects/op-tee/, 2023.

[7] Peach. https://peachtech.gitlab.io/peach-fuzzer-community/, 2023.

[8] Titan-M. https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/, 2023.

[9] AKF. https://github.com/purseclab/AKF, 2024.

[10] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[11] Sebanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *Proceeding of the Information Security Theory and Practice (WISTP)*, 2018.

[12] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. Teezz: Fuzzing trusted applications on cots android devices. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[14] Guoyun Duan, Yuanzhi Fu, Boyang Zhang, Peiyao Deng, Jianhua Sun, Hao Chen, and Zhiwen Chen. Teefuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation. *Future Generation Computer Systems (FGCS)*, 2023.

[15] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[16] Chenlin Huang, Yusong Tan, Guoyun Duan, Zhiwen Chen, Boyang Zhang, Peiyao Deng, Qianxiang Zhang, Jianhua Sun, Hao Chen, Guoqing Xiao, et al. A coverage-guided fuzzing framework for trusted execution environments. In *Proceedings of the IEEE Int Conf on High Performance Computing & Communications; Data Science & Systems; Smart City; Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 2021.

[17] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[18] Abdullah Imran, Habiba Farrukh, Muhammad Ibrahim, Z Berkay Celik, and Antonio Bianchi. {SARA}: Secure android remote authorization. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2022.

[19] Paul Leignac, Olivier Potin, Jean-Baptiste Rigaud, Jean-Max Dutertre, and Simon Pontié. Comparison of side-channel leakage on rich and trusted execution environments. In *Proceedings of the Workshop on Cryptography and Security in Computing Systems (CS2)*, 2019.

[20] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[21] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. Reversing and fuzzing the google titan m chip. In *Proceedings of the Reversing and Offensive-oriented Trends Symposium (ROOTS)*, 2021.

[22] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[23] Dan Rosenberg. Reflections on trusting trustzone. *BlackHat USA*, 2014.

[24] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's {TrustZone} keymaster design. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2022.

[25] Di Shen. Exploiting trustzone on android. *BlackHat USA*, 2015.

[26] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In *Proceedings of the Annual International Cryptology Conference (Crypto)*, 2017.

[27] Jie Wang, Kun Sun, Lingguang Lei, Shengye Wan, Yuewu Wang, and Jiwu Jing. Cache-in-the-middle (citm) attacks: Manipulating sensitive data in isolated execution environments. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.