École Polytechnique Fédérale de Lausanne

TrustDymbex: Trusted Application Dynamic symbolic executor for vulnerability leveraging

by Louis Henri Daniel Pepito Dumas

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Expert Reviewer
External Expert

Marcel Busch
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 14, 2024

Abstract

Double fetch vulnerabilities pose significant risks in modern software systems. They occur when a value is fetched from a memory location twice, and the memory location is modified between the two fetches. This can lead to serious memory corruption and thus major security breaches. Today, mobile systems widely use Trusted Execution Environments and Trusted Applications as a safe way to perform critical computations. However, they are not immune to double fetch vulnerabilities. Prior work has generally focused on kernel vulnerabilities and has not addressed trusted mobile applications adequately. Since Trusted Applications are increasingly used, it is urgent to propose tools to detect such bugs. We propose trustDymbex: Trusted Application Dynamic Symbolic Executor, a tool that could leverage known suspicious operations to detect double fetch vulnerabilities in Trusted Applications more precisely and generate inputs that could be used in real execution. We test our program with several test cases to assess the design functionality and real cases to leverage double fetch bugs based on some suspicious addresses.

1 Introduction

User requirements for security reach new heights every day. Complex systems are widely distributed and highly connected, especially mobile devices. This is why it is crucial to have a secure environment to perform critical operations. This is why Trusted Execution Environments (TEEs) are used. They are supposed to provide a secure environment to run critical code. This critical code is called Trusted Application (TA). TAs work in a similar manner as system calls. By design there is a difference of trust between user space and the memory space used by the Trusted Applications . Due to the modern multicore architecture, current systems are highly sensitive to race conditions. A classic class of bugs that violates system security is the double fetch bugs. They occur when a value is fetched from a memory location twice, and the data in memory is modified between the two fetches. This can lead to inconsistent states and unexpected behavior and thus major security breaches, especially if the fecthes appear in a high-privileged process.

Today, mobile systems widely use Trusted Execution Environments and Trusted Applications as a safe way to perform critical computations. However, they are not immune to double fetch bugs. Knowing the locations of suspicious fetches and uses, we want to determine if these operations are reachable in order, the inputs constraints to get on their path and generate inputs that could be used in real execution to confirm the presence of double fetch bugs in Trusted Applications.

Previous research has primarily concentrated on kernel vulnerabilities and has overlooked addressing Trusted Applications on mobile systems thoroughly, even though similar methods could be used in this context. Since Trusted Applications are increasingly used, it is urgent to propose tools to detect such bugs. trustDymbex is an approach to 1) verify reachability of potential double fetches and 2) generate TA-specific concrete test inputs to execute this path, a tool that could refine the results of static analyzers to get precise results and generate inputs that could be used in real execution. We will present some experiments and results that assess the efficacy of our tool by testing it in several test cases.

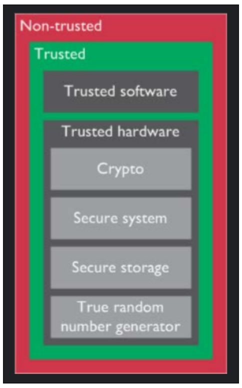
2 Background

Trusted Execution Environement and Trusted Applications

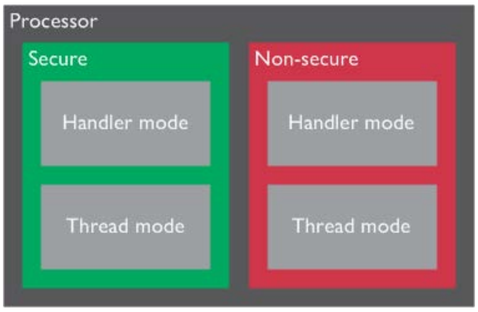
Trusted Execution Environments (TEEs) are special areas in a computer or mobile device that run sensitive tasks separately from the rest of the system as in Figure 1a. They ensure that the code running inside them is genuine, keeps important data safe, and prevents tampering. TEEs can prove to other systems or parties that they are secure and can be updated safely. They are designed to block all software attacks and even some physical attacks on the device's memory [8].

A Trusted Application is an application running inside a TEE. Since TEEs are supposed to be secure, they are used to run critical code such as payment or cryptographic code. Therefore, it is crucial to ensure that they are free of critical vulnerabilities.

Mobile devices widely use processors based on the ARM architecture, and specifically, we are interested in Cortex-A processors. In this architecture, TEEs are implemented by ARM TrustZone technology [1]. In ARM, TEE is implemented at the processor level can execute instruction in two modes: the *secure state* that have access to the whole memory while in *non-secure* it can only access to System registertha allow non-secure accesses (see Figure 1b) state



(a) Secure world components



(b) TrustZone secure and non-secure states at processor level and Secure world component

This technology is employing exception levels (ELs). *EL0* is utilized for all unprivileged executions, *EL1* serves as the privileged mode used by the kernel, *EL2* is for executing the Hypervisor (see Figure 2), which is not relevant to us, and *EL3* is dedicated to executing the *Secure monitor*. For

Cortex-A processors, the *Secure monitor* serves as the sole entry point to use TrustZone technology. It is responsible for switching between the normal world (NW) and the secure world (SW). The secure world can execute code at *EL0*, *EL1* and *EL3* in a *secure state*. All code running in the secure world is considered trusted and has access to the complete memory space (secure and non-Secure) and at *EL3* it can access all the system control resources.

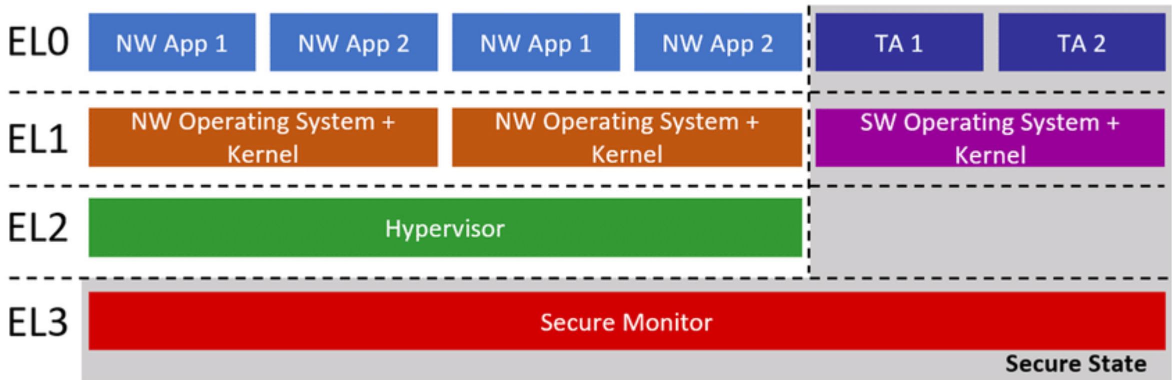


Figure 2: TrustZone organization with the different exception levels, what component are running and in which state

The *Secure monitor* is used to make calls to the secure OS, itself is interracting with TAs. The Figure 3 shows how these elements are interacting.

Shared memory in Trusted Applications

Usually, when a user wants to perform a critical operation, it makes a system call. They then pass the necessary parameters to the kernel: values, a pointer to shared memory, etc... Interactions between the user space and TAs are very similar.

The user application passes the necessary parameters to the TA through shared memory. the necessary parameters to the Trusted Application through shared memory through a TEE driver that is responsible for passing the parameters to the Secure world OS through the *Secure monitor* which will pass them to the TA.

Double fetches

Modern processors heavily rely on multi-core architecture, leading to the design of multi-threaded applications. This trend is evident across all types of devices, including mobile devices. While multi-threading can significantly enhance application performance, it also introduces new types of bugs, particularly race conditions resulting from concurrent access to shared resources.

We are particularly interested in a specific type of race condition known as Time-Of-Check-Time-Of-Use (TOCTOU) bugs, specifically double fetch bugs. During execution, it happens that some data are fetched and checked. Thus, the data are supposed to be constrained by some conditions. Double

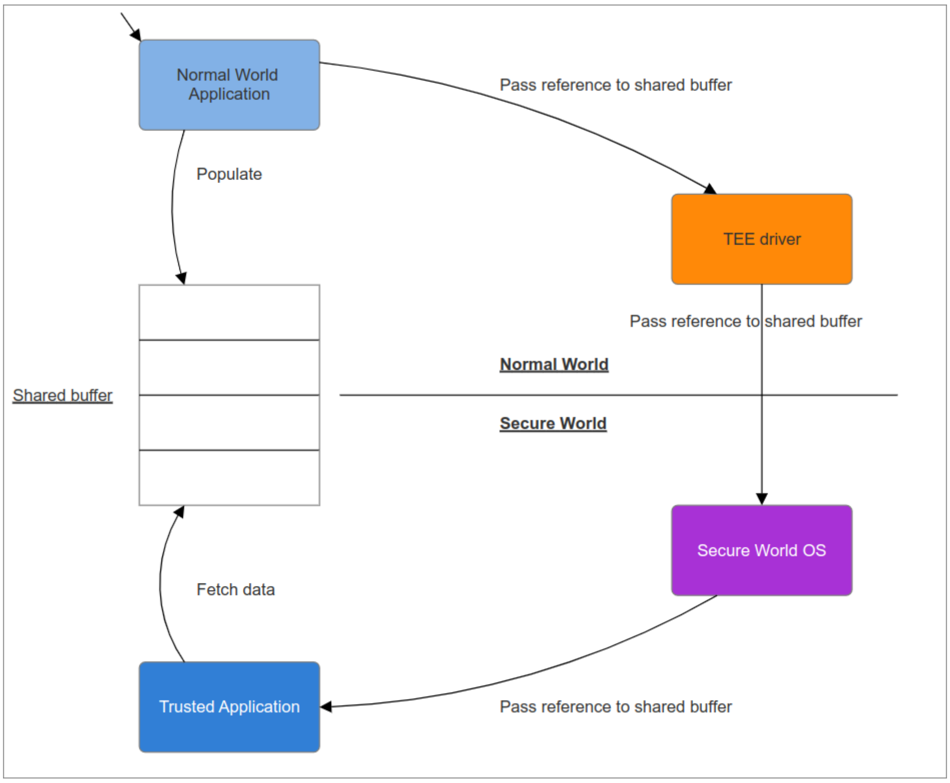


Figure 3: Shared memory scheme

fetch bugs appear when the data is fetched again and used without rechecking the conditions, allowing the possibility of modification between the two fetches.

In our scenario, referring to Figure 2, we have a thread running a normal world app at *EL0* and concurrently we have a thread running a Trusted Application at *EL0* but in *secure state*. Information is passed between the two threads through shared memory. The double fetch bug occurs when the TA fetches the data twice and the normal world app modifies the data between the two fetches.

The world of symbolic testing

There are multiple analysis techniques that use symbols to study program behavior. The relevant ones are abstract interpretation, symbolic execution, dynamic symbolic execution, underconstrainted dynamic symbolic execution and concolic execution.

Abstract interpretation Abstract interpretation is a static analysis technique that approximates program behavior by formalizing programming language semantics by mapping them to mathematical objects, and then approximates the values of variables by using abstract domains. For example, instead of using exact values, you might approximate variables by considering only their signs or a range of values. By doing so, you can study and search for unexpected states that can be reached through these approximations[7].

Symbolic execution Symbolic execution is a dynamic analysis technique that consists of executing a program in a abstract way. Instead of using concrete values, it uses symbols to represent arbitrary values. Each time the execution encounter statement that apply constraints on values (like a conditional statement), it creates a logical formula (symbolic expression) that represents the possible values that can be used to reach this statement. We can then extract concrete values from the constraints collected during the symbolic execution.

Dynamic symbolic execution Dynamic symbolic execution (DSE), uses an emulated environment to execute the program. It will track register and memory states at any given time, which is not done by symbolic execution. When the execution encounters a branch, it will follow both path[12].

Underconstrainted dynamic symbolic execution Underconstrainted dynamic symbolic execution is a subcategory of dynamic symbolic execution that aims to avoid the state explosion problem. It is able to start the execution at any function of the program and has symbolic values that do not care abpit previous constraints that they should have encountered in a normal symbolic execution[12].

Concolic execution Concolic execution is a combination of concrete and symbolic execution. The symbolic execution engine runs in parallel with the concrete engine. Each time a branch is encountered, the symbolic execution will register the constraint[11]. This method is limited to the initial input you gave to the program. It will collect constraints only for the one encounter during the execution. If some constraints are on an unexplored path, they won't be registered.

In this project, we study TAs outside of their normal environment. We work with executable files and TAs execution strongly depends on interactions with the normal world. We don't have the intrumentation capabilities to run the TA with concrete value efficiently. Moreover, TAs could have multiple entrypoints[4], which makes it even more difficult to know where to start the execution and what are data supposed to look like at each entrypoints. This is why our approach will use Underconstrainted dynamic symbolic execution to study the TAs behavior.

3 Motivation

Even if a lot of work has been done on double fetch bugs in kernels (see section 9), It seems that little work has been done on Trusted Applications. At most, there are some exploitation-oriented blog posts [9] that are focused on generic TA exploitation and not always focused on double fetches. The main objective of this work is therefore to fill part of this gap and explore ways to find these bugs in that context. The main challenge of this project is to detect double fetch vulnerabilities in TAs by symbolically executing way in a efficient way to avoid all unecessary code execution. Another important challenge is to be able to discriminate between benign double fetches and suspicious ones.

4 Threat Model

Our work is focused on Trusted Applications. By design, Trusted Applications are built to be isolated from the user space and the kernel space (*EL0* and *EL1*) (see section 2 for more details). Thus, it would be realistic to consider an adversary with full control over the kernel and the user space who can instantiate any Trusted Application and manipulate the shared memory. Nevertheless, our work is especially focused on the manipulation of the shared memory between the user space and the Trusted Application (*EL0* in *non-secure state* and *secure state*), Figure 3 is describing how normal world app and TA are interacting with the shared memory. So it is sufficient to consider an adversary that only has control over the user space. The adversary's goal is to manipulate the shared memory in such a way that it would result in arbitrary code execution in *secure state* and thus gaining control on the full memory space.

5 Design

The aim of trustDymbex is to leverage the addresses of suspicious fetches and uses operations to confirm the presence of double bugs in Trusted Applications. It works on a two-stage process: first, we recover the constraints to reach the potential double fetch, then we execute the binary with concrete values to try to assess the reachability of the path. We start the execution after at the function that receives the reference to the populated shared buffer (see section 2). In ARM architecture, the calling convention sets the arguments to be in the four first registers. The registers that are holding values are set to be symbolic. For those that are holding addresses, we set them to be a concrete address that points to some free space in memory. The shared buffer is instantiated only with symbolic values.

To ensure the success of this process, we have to take into account the following design decisions: the code traversal policy, how we eliminate false positive double fetches and how we generate inputs that can reach the last use.

Let's take a concrete example to illustrate how trustDymbex should act.

```
TEE_Result TA_InvokeCommandEntryPoint(void __maybe_unused *sess_ctx,
                        uint32_t cmd_id,
                        uint32_t param_types, TEE_Param params[4])
{
  (void)&sess_ctx; /* Unused parameter */
  size_t tmp_len;
  msg_t m;
  TEE_Param param;
  char* buf;

  switch (cmd_id) {
    case TA_HELLO_WORLD_CMD_INC_VALUE:

      // we're fetching it once
      tmp_len = *(size_t*)params[0].memref.buffer;

      if (tmp_len > MAX_BUF_LEN) {
        return TEE_ERROR_BAD_PARAMETERS;
```

```
19      }
20
21      // Make a call without tainted parameter
22      int arr[] = {1, 2, 3, 4, 5};
23      int* tmp_buf = arr;
24
25      fun(tmp_buf);
26      // Make a call to a function with tainted parameter
27      fun(params[0].memref.buffer);
28
29      // we're fetching it twice
30      tmp_len = *(size_t*)params[0].memref.buffer;
31
32      // this overflow shows if you're naughty or nice
33      TEE_MemMove(&m, params[0].memref.buffer, tmp_len);
34
35      return TEE_SUCCESS;
36
37    default:
38      return TEE_ERROR_BAD_PARAMETERS;
39    }
40    return TEE_SUCCESS;
41 }
```

Listing 1: TA test case

Supposing we want to run trustDymbex on the example Listing 1. We know the addresses corresponding to the fetch for line 15: `tmp_len = *(size_t *)params[0].memref.buffer;`, the use at line 17 `if(tmp_len > MAX_BUF_LEN)\{`, the second fetch at line 27 `tmp_len = *(size_t *)params[0].memref.buffer;` and the second use at line 30 `TEE_MemMove(&m, params[0].memref.buffer, tmp_len);`. trustDymbex should be able to symbolically execute the TA, asses that the operations appear in order, that the very same location is fetched twice, recover that `cmd_id` is equal to `TA_HELLO_WORLD_CMD_INC_VALUE`, which bytes in the shared buffer should be less than `MAX_BUF_LEN` and generate concrete inputs consistants with the constraints.

Algorithm 1 Reachable

Require: $listOfState$ a list of states, $currentState$ the current state
  for state in $listOfState$ **do**
    if state is in the history of $currentState$ **then**
      return true
    end if
  end for
  return false

Code traversal policy

As explained in section 3, we want to have an efficient symbolic execution. To do so, we need to be specific about how we traverse the code. For this, we use a combination of guided execution and taint tracking. We start with four addresses corresponding to two fetch operations and two uses.

These operations appear in the following order: the first fetch, the first use, the second fetch, and the second use.

At first, we want to recover all the constraints that make it possible to reach the second use by passing through all the other operations. To recover these constraints, we first make all shared memory between the user and the trusted space symbolic, as well as anything that can affect the control flow of the Trusted Application. This way, we will be able to branch at every necessary step to reach the first fetch.

Next, we want to be efficient we have to take into account that during execution, we can encounter some function calls that will do nothing interesting in the memory or any important parameter. To skip their execution, we taint the shared memory and check for each function call if the parameters are tainted. If they are not, we skip the function execution. If we go back to the example at Listing 1, we will skip the function `fun` at line 23 but not at line 27. We also add an *inter-function level*, inspired by *BootStomp*[6]. We keep track of the function deepness and if a function call is supposed to make the function deepness exceed the inter-function level then we skip the function execution.

False positive elimination

A double fetch occurs if the same address is fetched twice. It is very plausible that at the second fetch, the value has been relocated somewhere else in the memory and thus it won't be a "real" double fetch. To assert that the double fetch is a not a false positive, we recover the exact addresses that are fetched and compare them. If they are the same, we consider it as a true-positive double fetch.

Input generation

Now that we have the constraints for a given valid double fetch, we will try to generate inputs that will fit the constraints. It is a trivial step since we have the constraints; we can just generate random values that will fit the constraints.

6 Implementation

TrustDymbex is implemented in Python 3.12.3 using Angr 9.2.105[12] platform and every execution was done on Trusted Applications compiled for the ARM architecture using the Global Platform's *TEE Internal Core API Specification*[13]. The following explains how we implemented the design decisions using the different tools proposed by Angr.

Code traversal policy

First, we need to recover the constraints that make it possible to reach the second use by passing through all the other operations provided by the static analyzer. For this, we use the `SimulationManager` object provided by Angr. For each operation we set the address of the operation as the *target* and

Algorithm 2 trustDymbex procedure

Require: symbolic memory and inputs ready, $inter-function\ level = n$
  $listOfState \leftarrow []$
  for fetches and uses addresses **do**
    if address in the same basic block as the previous one **then**
      append the state to $listOfState$
      pass to the next address
    end if
    execute until we reach a function call or the address
    if we reach the address **then**
      if $Reachable(listOfState, currentState)$ **then**
        append the state
      else
        return
      end if
    else if we reach a function call **then**
      if tainted parameters or $deepness > inter-function\ level$ **then**
        skip the function
      end if
    end if
  end for
  $fetch1 \leftarrow listOfState[0]$
  $fetch2 \leftarrow listOfState[2]$
  if $fetch1.address \neq fetch2.address$ **then**
    return
  end if
  generate input from $listOfState$

the address of the next operation as the *target*. We then step until we reach the target address before targeting the next operation. Each time we encounter a state containing the address of the operation we are looking for, we store it in a list. Before trying to get to the next target we verify that the previous collected `SimState` are in the history of the one we reach to assert that the path exists.

To skip the execution of functions that do not take tainted parameters, we use the `SimInspector` object. We set a breakpoint for each function call. If the parameters are not tainted we use a `SimProcedure` which we called `generic_hook` that will just skip the execution of the function. If the parameters are tainted, we continue the execution. Note that for external calls that use tainted parameters we need to create excutable dependent `SimProcedures` that will handle the call.

To taint the parameters, we create a class `Taint` based on claripy `Annotations` and verify on each parameter if there is a `Taint` object in there annotation list. For the sake of the execution, all parameter should be tainted and if they are pointer they should also be tainted as well as all the value they contains. Unfortunately, Angr does not seems to be able to keep the taint for concrete bitvector when they are passed directly to a register and not store in memory first.

TAs often use their own structure with type aliases to pass parameters. Angr know only about built-in types so we have to parse and register the new types by hand and know about what the TA is expected.

Hooking

Some function are call to external libraries and thus the code is not available to us. To handle these calls, we use the `SimProcedure` object that make us able to hook the call and execute our own code. We have to create a `SimProcedure` for each external call that we want to handle and thus this is very binary dependent.

False positive elimination

To eliminate false positives, we use the same tools as in section 5 to skip function execution. When we reach a state that contains a target that correponds to a fetch operation, we use the `SimInspector` to install a breakpoint on the fetch operation. We use the breakpoint to collect which address in memory has been fetched.

When we have collected all the necessary states, we compare the fetched addresses as described in section 5. We stop the execution if we conclude that the double fetch is a false positive.

Input generation

Angr provides an easy way to generate input that fits the constraints. From the state corresponding to the last use, we can use the `state.solver` object to generate random values that fit the constraints. All these values will be stored in a file to be used in the real execution. For inputs that are value in the shared buffer we consider them in a big-endian fashion and consider the first byte to be the least significant one.

7 Evaluation

To assert that trustDymbex is working as intended we will run the following experiments. We first wrote test cases TAs containing double fetch bugs. These TAs have different code for testing the different features of trustDymbex. Secondly, we use different real TAs with known addresses of fetch and uses to assess the correct behavior of trustDymbex. In addition, we use some known double fetch bugs[4] to verify that trustDymbex can leverage them. All experiences were ran on docker environment with a Ubuntu 22.04 image and the Angr 9.2.105 platform with an *inter-function level* set to 1.

Test cases

All test cases were compiled for the ARM architecture using the Global Platform's *TEE Internal Core API Specification*[13]. We did in total 10 test cases. They were designed to test the different features of trustDymbex as automatically as possible. They have the nomenclature format `xxx-sharecase`, where `xxx` is the number of the test case. Note that as we cross-compiled the test cases ourselves, in these cases all call to external libraries are in the binary which won't be the case for these real cases and thus Hooking only concerns real cases. We notice a strange behavior in all test cases that involve a function call. The function executed at the breakpoint is called multiple times even if the function is called only once. We didn't manage to understand why this is happening and we suspect these mutliple executions to be responsible for a major perfomance loss. Our supposition is that has to do with how breakpoint are implemented by the `SimInspector`. There is an ongoing discussion about this issue on the Angr GitHub[14].

Code traversal and input generation: 000-sharecase, 001-sharecase, 005-sharecase, 009-sharecase and 010-sharecase

000-sharecare is the simplest one. It does simple and succesive fetch and uses according to what a double fetch is supposed to look like. It is used to test the trivial traversal policy and basic input generation. We are supposed to recover the followings constraints: `cmd_id` is equal to 0 and some bytes in the share buffer are less than 128.

001-sharecare is very similar to 000-sharecases. It does the same succession of fetch and uses but with a different syntax. It appear to have the exact same result has 000-sharecases.

005-sharecase test adds conditions on the `param_types`. It is used to assert the constraints recovery and the input generation. The test case passes as expected. We could build some more complexe test cases to be more exhaustive about the input generation.

009-sharecase is a modified version of 000-sharecase that add a for loop between the first use and the second fetch. It is used to assert that the code traversal policy is working correctly. The test pass as expected.

010-sharecase is a test that make nested function call. The entrypoint make a call to a function `fun1` that make a call to another function `fun2`. It is used to assert the correct behavior of the *inter-function level*. The test pass as expected.

Taint verification and Hooking: 002-sharecase

This test uses a library call to `TEE_MemMove`. It is used to test the taint verification, the call handling and hooking. The text pass as expected. However, on test cases library function are present in the binary so it is not to test the hooking properly. We should write a test case in which the library function are not in the binary.

Taint verification: 003-sharecase, 004-sharecase, 007-sharecase

003-sharecare use an intern call to `fun`. This is used to test the taint verification and assert that double fetch are correctly handled when they appear during an function call. The test pass as expected.

004-sharecare uses multiple intern call to `fun` with random integers. It is used to assert that the taint verification is working correctly when no parameter are tainted. We test for different number of parameter to very how the taint verification behave depending on how much register are used and if some parameter are on the stack. All call are skipped as expected.

007-sharecare use an intern call to `fun` first with untaited parameters and then with tainted parameters. It should assert that the taint verification is working correctly and that function are hooked and unhooked correctly. The test pass as expected.

008-sharecare is a test case that is similar to the other test but it has only one parameter on the stack. It is used to assert that the taint verification is working correctly when the parameter are on the stack. The test pass as expected.

Interprocedural double fetch: 006-sharecase

This test is a bit special it has the first fetch and use in the `TA_InvokeCommandEntryPoint` and the second fetch and use in a function call. This test was designed to study how trustDymbex behave during interprocedural double fetches. The test pass, so trustDymbex is able to handle interprocedural double fetches.

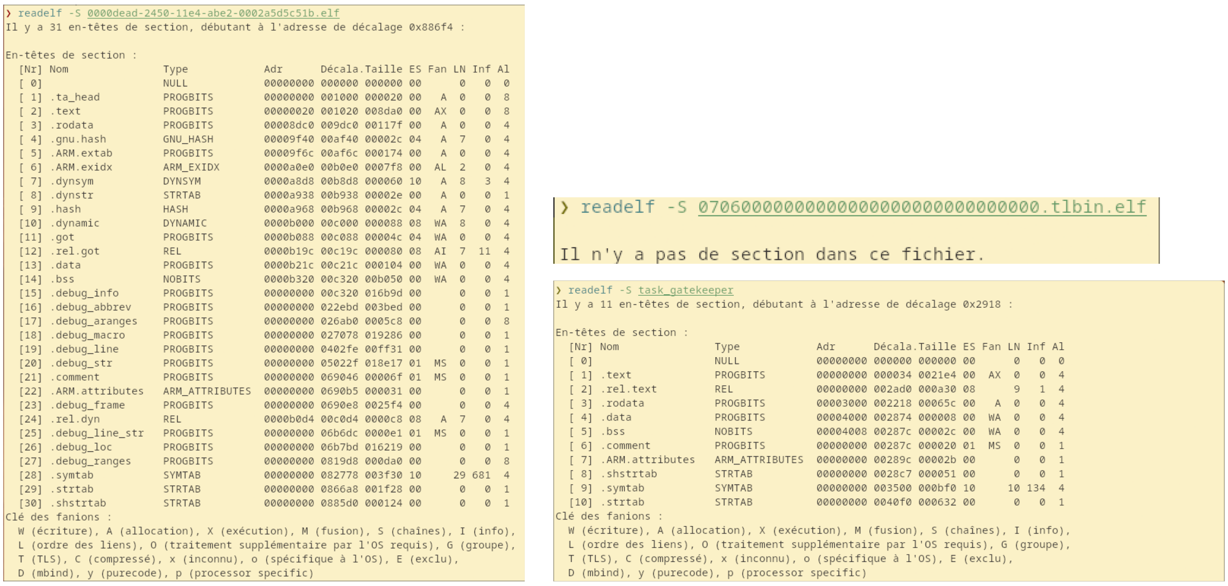
Real cases

To test trustDymbex on real cases, we used TAs extracted from real devices. Specifically from the Huawei P9 Lite and the Samsung Galaxy S6 Edge. The first one implements TrustZone using the GlobalPlatform API and the second one uses Kibini. We have three test cases for each device. For more details about these TAs see [4]. With the current design of trustDymbex, we are running a new simulation for each quadruplet of fetch and uses addresses, this is not efficient and lead to excessively long running time for the real cases.

Huawei P9 Lite

For the Huawei P9 Lite, we have `task_storage`, `task_keymaster` and `task_gatekeeper`, respectively responsible for implementing an ecrypted file system, cryptographic key managment and password authentication. We encounter different challenges with these TAs. The main one is that the TAs does have missing meta data in their headers. First, the relocation are not specified (see Figure 7). This lead to a problem, we know the addresses of fetches and uses for `task_keymaster` but they were collected using Ghidra. However, the way Ghidra handle the default value for the relocation is not the same as the one used by Angr. Thus we have to recompute the addresses of fetches and this is binary dependent as Ghidra does necessarily not use the same base address and the same offsets.

To solve this problem we wrote a script that will recompute the addresses of fetches and uses. We compute the offset in between the beginning of the function and the address of the fetch and uses, then use these offset along with the adress in the function in Angr to have the correct address.



```
❯ readelf -S 0000dead-2450-11e4-abe2-0002a5d5c51b.elf
Il y a 31 en-têtes de section, débutant à l'adresse de décalage 0x886f4 :

En-têtes de section :
  [Nr] Nom               Type            Adr      Décala.Taille ES Fan LN Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .ta_head          PROGBITS        00000000 001000 000020 00   A  0   0  8
  [ 2] .text             PROGBITS        00000020 001020 008da0 00  AX  0   0  8
  [ 3] .rodata           PROGBITS        00008dc0 009dc0 00117f 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        00009f40 00af40 00002c 04   A  7   0  4
  [ 5] .ARM.extab        PROGBITS        00009f6c 00af6c 000174 00   A  0   0  4
  [ 6] .ARM.exidx        ARM_EXIDX       0000a0e0 00b0e0 0007f8 00  AL  2   0  4
  [ 7] .dynsym           DYNSYM          0000a8d8 00b8d8 000060 10   A  8   3  4
  [ 8] .dynstr           STRTAB          0000a938 00b938 00002e 00   A  0   0  1
  [ 9] .hash             HASH            0000a968 00b968 00002c 04   A  7   0  4
  [10] .dynamic          DYNAMIC         0000b000 00c000 000088 08  WA  8   0  4
  [11] .got              PROGBITS        0000b088 00c088 00004c 04  WA  0   0  4
  [12] .rel.got          REL             0000b19c 00c19c 000080 08  AI  7  11  4
  [13] .data             PROGBITS        0000b21c 00c21c 000104 00  WA  0   0  4
  [14] .bss              NOBITS          0000b320 00c320 00b050 00  WA  0   0  4
  [15] .debug_info       PROGBITS        00000000 00c320 016b9d 00      0   0  1
  [16] .debug_abbrev     PROGBITS        00000000 022ebd 003bed 00      0   0  1
  [17] .debug_aranges    PROGBITS        00000000 026ab0 0005c8 00      0   0  8
  [18] .debug_macro      PROGBITS        00000000 027078 019286 00      0   0  1
  [19] .debug_line       PROGBITS        00000000 0402fe 00ff31 00      0   0  1
  [20] .debug_str        PROGBITS        00000000 05022f 018e17 01  MS  0   0  1
  [21] .comment          PROGBITS        00000000 069046 00006f 01  MS  0   0  1
  [22] .ARM.attributes   ARM_ATTRIBUTES  00000000 0690b5 000031 00      0   0  1
  [23] .debug_frame      PROGBITS        00000000 0690e8 0025f4 00      0   0  4
  [24] .rel.dyn          REL             0000b0d4 00c0d4 0000c8 08   A  7   0  4
  [25] .debug_line_str   PROGBITS        00000000 06b6dc 0000e1 01  MS  0   0  1
  [26] .debug_loc        PROGBITS        00000000 06b7bd 016219 00      0   0  1
  [27] .debug_ranges     PROGBITS        00000000 0819d8 000da0 00      0   0  8
  [28] .symtab           SYMTAB          00000000 082778 003f30 10     29 681  4
  [29] .strtab           STRTAB          00000000 0866a8 001f28 00      0   0  1
  [30] .shstrtab         STRTAB          00000000 0885d0 000124 00      0   0  1
Clé des fanions :
  W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
  L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe),
  T (TLS), C (compressé), x (inconnu), o (spécifique à l'OS), E (exclu),
  D (mbind), y (purecode), p (processor specific)
```

```
❯ readelf -S 07060000000000000000000000000000.tlbin.elf

Il n'y a pas de section dans ce fichier.
```

```
❯ readelf -S task_gatekeeper
Il y a 11 en-têtes de section, débutant à l'adresse de décalage 0x2918 :

En-têtes de section :
  [Nr] Nom               Type            Adr      Décala.Taille ES Fan LN Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 0021e4 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 002ad0 000a30 08      9   1  4
  [ 3] .rodata           PROGBITS        00003000 002218 00065c 00   A  0   0  4
  [ 4] .data             PROGBITS        00004000 002874 000008 00  WA  0   0  4
  [ 5] .bss              NOBITS          00004008 00287c 00002c 00  WA  0   0  4
  [ 6] .comment          PROGBITS        00000000 00287c 000020 01  MS  0   0  1
  [ 7] .ARM.attributes   ARM_ATTRIBUTES  00000000 00289c 00002b 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 0028c7 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 003500 000bf0 10     10 134  4
  [10] .strtab           STRTAB          00000000 0040f0 000632 00      0   0  1
Clé des fanions :
  W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
  L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe),
  T (TLS), C (compressé), x (inconnu), o (spécifique à l'OS), E (exclu),
  D (mbind), y (purecode), p (processor specific)
```

Figure 4: readelf -S output comparison

Secondly, some function are not recognised as function, so instead of creating breakpoint on the call we have to create breakpoint on each instruction and verify that it is a `bl` instruction which certainly occur a loss of performance.

We had log file containing the address of the fetches and uses for `task_keymaster`, for `task_storage` and `task_gatekeeper` we had to refer to *Finding Race Conditions in Trusted Applications*[4].

task_keymaster For this TA we have a log file containing different addresses of fetches and uses along with the name of the function in which they appear. Those functions are: `AuthTokenMatches.constprop.2`, `km_export_key`, `authentication_key`, `get_key_param`, `km_begin`, `generate_keynode`, `is_key_param_suport`, `parser_symmetric_keymaterial` and `km_get_key_characteristics` Due to the loss of performance, each simulation on this TA took more approximately 2 min with a *inter-function level* set to 1. By analyzing the binary using Ghidra, we were able to figure out which functions were called at which deepness. To collect result in a short amount of time we decided to avoid suspicious address that were not in function at level 1 and focus on only one function. We choose `km_export_key` as it was the function that has the smallest number of suspicious addresses (i.e 116). We measure the performance of trustDymbex by analyzing its running time as a function of the number of suspicious quadruplet. We try different path exploration, DFS and BFS, the exploration method according to Figure 5 does not seems to have an impact on the running time. With the current algorithm we start a new simulation for each quadruplet so we expected long runtime but proportionally to the number of suspicious quadruplet. Here it seems that the running time is growing exponentially, we didn't find an explanation for this. Notwithstanding, we confirm 5 double fetch it this function.

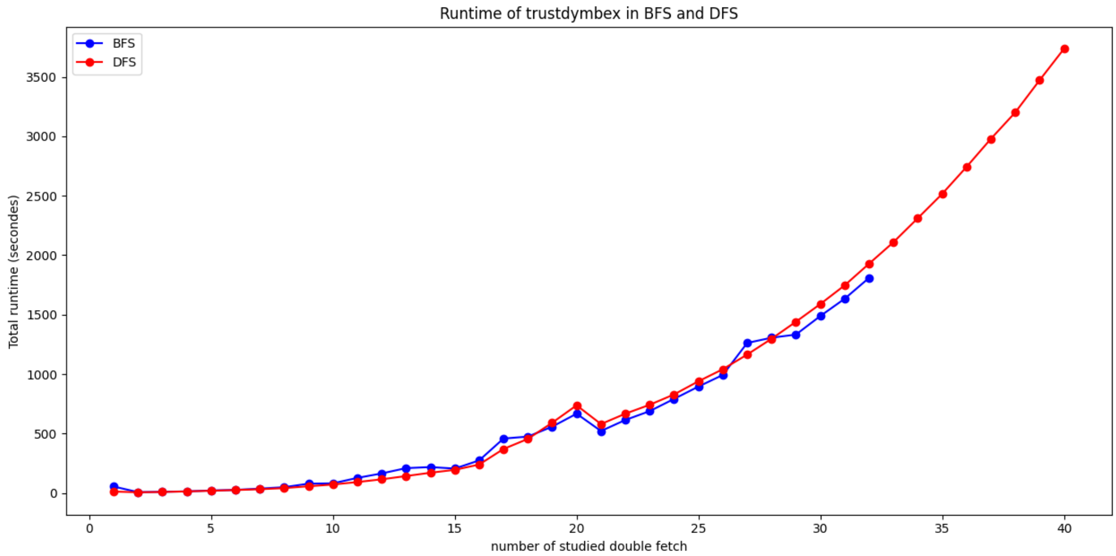


Figure 5: Running time of trustDymbex on task_keymaster

Samsung Galaxy S6 Edge

For the Samsung Galaxy S6 Edge, we have `TEE_Keymaster`, `TRUST_Keymaster` they both have the same purpose as `task_keymaster` and `HDCP` that is used to encrypt media data[4]. The binary correponding to these TAs are respectively `07060000000000000000000000000000.tlbin`, `ffffffff00000000000000000000003e.tlbin` and `ffffffff000000000000000000000005.tlbin` . These files are in mclf format and we have to convert them to elf format to be able to use them with Angr. We used a converter[5] to convert them to elf format. The resulting ELF does not have any header as describe in Figure 7 leading to the same kind of problems as for the Huawei P9 Lite. Moreover, those TAs do not use the GlobalPlatform API and thus the implementation should be modified accordingly with another symbolic initialization. With all these constraint and too little time, we decided to stay focus on the Huawei P9 Lite.

8 Limitations

One of our theorical limitations is that we limit valid double fetches to the exact same address. It is very possible that a first fetch loads a range of addresses in the shared buffer and that a second one fetch loads another range of address that do not start at the same address but the corresponding memory area are overlapping. In this case verifying that a double fetch is not false positive would be more complicated. We also already mentionned the poor running perfomance of the tool that make it difficult to run on real cases.

9 Related Work

For large code bases, manual checking is no longer an option, so numerous techniques for the automatic detection of double fetches have been proposed over time. These techniques mainly focus on kernel vulnerabilities.

A common approach is to use a combination of static and dynamic analysis to detect double fetches. For example, Meng Xu et al. created a tool called *Deadline* [16], which statically analyzes kernels to find multi-reads and uses symbolic checking to confirm the existence of bugs. They also provided formal definitions of double fetches. Those formal definitions were essential to our work as a rigorous basis to think about double fetches and especially theorize about the limitations of trustDymbex. The path construction of *Deadline* is really close to our approach in his idea to reconstruct the path to the double fetch. Nevertheless, one of the goal of trustDymbex is also to generate inputs that can reach the last use which is not at all taken into account in *Deadline*.

Another approach is to use dynamic analysis at the CPU-operation level, examining memory and cache access patterns. M. Schwarz et al.[10] developed a kernel fuzzer along with a side-channel cache attack to detect double fetches. Similarly, Mateusz Jurczyk and Gynvael Coldwind created *Bochspwn*[3], a tool that monitors memory accesses to intercept memory references and check for the same fetched locations. Those tools have are very performant in their task but they are not adapted to Trusted Applications. Indeed, due to the nature of the TA we cannot run outside a real environment thus dynamic analysis is not an option for us or will ask a lot of instrumentation, even if memory pattern analysis could be an approach using Angr and its memory emulation capabilities.

Creating tools that automatically fix or mitigate double-fetch bugs is also an intensive field of research. For example, the Midas tool [2] automatically fixes double fetches in the kernel by snapshotting objects at the first access and maintaining unique copies during syscalls for subsequent reads. M. Schwarz [10], in addition to their fuzzer, used hardware transactional memory to eliminate double fetch bugs. P. Wang et al. [15] used the Coccinelle engine and pattern analysis to automatically change kernel code with static analysis.

Regarding symbolic execution and state explosion mitigation, the *BootStomp* tool [6] uses dynamic symbolic execution to detect bootloader vulnerabilities. Their taint tracking and code traversal policies serve as inspiration for our tool. They check if parameters are tainted before executing the function, and if not, they step out of it.

10 Future Work

For the traint on concrete bitvector, we could eventually hold a static map that contain the bitvector and a boolean that indicate if it is tainted or not. We could improve the perfomance of trustDymbex by minimizing the number of emulation. To keep the information of the function we are currently analyzing we could imagine to make a emulation that will try to find all associated double fetches in one one run. Creating a dataStructure that would hold the information about the progress of finding basic blocks and the associated double fetches could be a good idea. We would when all the double fetches are found or until the execution finish. Another idea could be to target the funciton we are studying, if we know its name, and be even more precise in our guided symbolic execution.

11 Conclusion

We have presented trustDymbex, a tool that uses symbolic execution to detect double fetch vulnerabilities in Trusted Applications based on the approach of other tool such as *BootStomp*. Unfortunately, we didn't manage to implemeents a functioning version of trustDymbex on real cases even if we provied test cases to work around the different features of the tool. Still, trustDymbex perform correct result on simple cases that does not involve function calling. We finally insist that the design seems good and that it could be a good tool to detect double fetches in TAs if the taint tracking is fixed.

Bibliography

[1] *ARM documentation.* URL: https://developer.arm.com/documentation/.

[2] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. "Midas: Systematic Kernel TOCTTOU Protection". In: *31st USENIX Security Symposium (USENIX Security 22).* Boston, MA: USENIX Association, Aug. 2022, pp. 107–124. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/bhattacharyya.

[3] Mateusz Jurczyk and Gynvael Coldwind. "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns". In: *Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns.* 2013, p. 69.

[4] Philipp Panzer. *Finding Race Conditions in Trusted Applications.* 2020.

[5] Quarskslab. *samsung-trustzone-research.* URL: https://github.com/quarkslab/samsung-trustzone-research/blob/master/tainting/mclf2elf/.

[6] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "BootStomp: On the Security of Bootloaders in Mobile Devices". In: *26th USENIX Security Symposium (USENIX Security 17).* Vancouver, BC: USENIX Association, Aug. 2017, pp. 781–798. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini.

[7] Mads Rosendahl. *Introduction to Abstract Interpretation.* DIKU, Computer Science University of Copenhagen, 1995.

[8] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA.* Vol. 1. 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[9] Eloi Sanfelix. *TEE Exploitation on Samsung Exynos devices.* 2019. URL: https://labs.bluefrostsecurity.de/blog/2019/05/27/tee-exploitation-on-samsung-exynos-devices-introduction/.

[10] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. *Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features.* 2017. arXiv: 1711.01254 [cs.CR].

[11] Koushik Sen. "Concolic testing". In: *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering.* 2007, pp. 571–572.

[12] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy.* 2016.

[13] GlobalPlatform Technology. *Tee internal core api specification version 1.1.2.50 (target v1.2), 2018.* URL: `https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf`.

[14] Voxanimus. *Project Angr issues.* URL: `https://github.com/angr/angr/issues/4682`.

[15] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. "How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel". In: *26th USENIX Security Symposium (USENIX Security 17).* Vancouver, BC: USENIX Association, Aug. 2017, pp. 1–16. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei`.

[16] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels". In: *2018 IEEE Symposium on Security and Privacy (SP).* 2018, pp. 661–678. DOI: `10.1109/SP.2018.00017`.