École Polytechnique Fédérale de Lausanne

# Leveraging Static Analysis on Binaries to Uncover Time-of-Check-Time-of-Use Bugs

by Pietro Moretto

## Semester Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Marcel Busch
Thesis Supervisor

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

February 6, 2024

# Abstract

This research explores the security of trusted applications, specifically focusing on Time-of-Check to Time-of-Use (TOCTOU) bugs within Trusted Execution Environments (TEEs). Trusted applications are crucial for tasks like digital payments, yet vulnerabilities persist, particularly in shared memory structures. Existing literature addresses TOCTOU bugs but falls short in the context of Trusted Applications (TAs), prompting the need for our work.

Our project adopts binary static analysis, leveraging Ghidra's P-Code intermediate representation, to uncover and address TOCTOU bugs in real-world proprietary trusted applications operating within ARM's TrustZone. The core idea involves initiating a data-flow analysis from a specified entry point function to detect potential double-fetch vulnerabilities within shared buffers.

Key results from our evaluation demonstrate the successful identification of various double-fetch patterns in different Trusted Applications, underscoring the effectiveness of our static analysis tool. For future work, we aim to test the static analyzer on a more extensive set of Trusted Applications, expand the supported functions and P-Code operations to increase the accuracy, and explore combinations with symbolic or concolic execution for enhanced bug detection.

# Contents

# Chapter 1

# Introduction

In the ever-evolving landscape of modern technology, Trusted Applications have become integral components in various domains, serving as essential tools for a diverse range of applications. Among these applications are security- and privacy-sensitive functionalities, such as digital payment and sensitive data processing, which are prevalent across different sectors. These applications operate within the context of a Trusted Execution Environment (TEE), a secure and isolated execution environment, designed to ensure the confidentiality and integrity of sensitive information and code. This isolated execution environment handles security-critical tasks, including the management of cryptographic keys or biometric data.

ARM's TrustZone, a set of hardware extensions, is a prevalent technology underlying the security architecture in various systems. It operates in two modes: the *non-secure Rich Execution Environment* (REE) and the aforementioned *secure Trusted Execution Environment* (TEE).

Despite the isolation mechanisms in place, the necessity for communication between the two environments introduces potential vulnerabilities. The shared memory infrastructure poses a specific risk: race conditions. Exploitation of these race conditions could compromise the isolation between the *Rich Execution Environment* and the *Trusted Execution Environment*, posing a significant threat to the security of the system.

This project addresses a specific type of vulnerability known as Time-of-Check to Time-of-Use (TOCTOU) bugs, which have the potential to lead to severe memory corruptions, and focuses on the detection of double-fetch bugs. Such memory corruptions, if exploited by adversaries, could enable them to compromise the affected system and gain full control.

Existing literature addresses TOCTOU bugs and race conditions detection on large programs [2, 14] and kernels [15], using both dynamic and static approaches. However, there remains a gap in the exploration of the topic of concurrency bugs in Trusted Applications, especially double-fetch vulnerabilities within the shared memory context of TEE implementations. There is one mention of

shared memory vulnerabilities in a talk given by Sanfelix [11], where he exploited a concurrency bug to gain arbitrary code execution in a Trusted Application.

The focus of this project is to adopt binary static analysis to identify and uncover TOCTOU bugs in proprietary real-world TAs. From a given entry point function, a data-flow analysis runs on the Ghidra's P-Code intermediate representation, in order to find potential double-fetches related to a specified shared buffer within the reachable code. By doing so, we aim to contribute to the enhancement of security measures, ensuring the robustness of TEEs and fortifying the defense against potential threats in this critical security domain.

# Chapter 2

# Background

## 2.1 Race conditions

The advent of the multi-core revolution has significantly impacted the landscape of computing, extending its reach to a diverse array of devices, including smartphones. The parallel execution of processes on multiple cores offers undeniable advantages, primarily in terms of performance gains. However, this parallelism introduces a new set of challenges, notably in the form of concurrency bugs. Among these, race conditions stand out as a critical concern. Race conditions occur when multiple threads concurrently access shared resources without proper synchronization, leading to unexpected behavior and security breaches. Concurrent access to shared resources, if not appropriately managed, can be exploited by malicious entities to compromise the integrity and security of a process. Many techniques have been studied to exploit a race condition. For example, a typical approach is to repeat a concurrency attack multiple times to increase the likelihood of the concurrency bug actually being triggered.

In the realm of kernels, which are highly concurrent processes, the risk of race conditions is particularly pronounced. Extensive research has been dedicated to the detection of race conditions in kernel environments [15]. One type of concurrency bug is *Time-Of-Check-Time-Of-Use* (TOCTOU) race conditions that often manifest as double-fetch bugs. *"Double-fetch bugs are a special type of race condition, where an unprivileged execution thread is able to change a memory location between the time-of-check and time-of-use of a privileged execution thread"* [12]. In the context of kernels, this vulnerability arises when a kernel thread reads from a user-space address multiple times, and a user thread modifies the data at that address between the fetches. A similar scenario happens in TEEs.

## 2.2 Double-fetches in TrustZone TEE

A Trusted Execution Environment (TEE) is a secure area of a processor that ensures the isolation of sensitive operations and data from the rest of the system. In the context of ARM devices, TrustZone technology is used to create this isolated and secure execution environment (TEE) alongside the Regular Execution Environment (REE), considered untrusted by design. REE is often referred to as the *Normal World*, where traditional computing tasks occur, including user applications and the kernel.

The architecture involves different Exception Levels (ELs) in ARM CPUs, each representing a specific privilege level. In the Normal World, user applications typically run in Exception Level 0 (EL0), while the kernel operates in Exception Level 1 (EL1).

The TEE introduces the concept of the *Secure World*, which operates in Secure Exception Levels, namely Secure EL0 (S-EL0) and Secure EL1 (S-EL1). The Trusted OS runs at S-EL1, providing an isolated environment for security-sensitive tasks. Trusted Applications (TAs) run in S-EL0 and offer services such as managing cryptographic keys or handling biometric data.

Communication between the Normal World (REE) and the Secure World (TEE) involves Client Applications (CAs) in the Normal World interacting with TAs in the Secure World. To facilitate this communication, a TEE driver is included in the Normal World's kernel. The transition between the worlds is managed by a Secure Monitor, typically operating at Exception Level 3 (EL3), through Secure Monitor Calls (SMCs). These SMCs ensure secure transitions and communication between the Normal World and the Secure World.

One critical aspect is the handling of data buffers passed between the Normal World and the Secure World. CAs in the Normal World create buffers with requests, and these buffers need to be securely transferred to TAs in the Secure World. The process involves the following steps:

- The CA populates a buffer with the request and passes it to the TEE driver via an ioctl invocation in the Normal World.
- The TEE driver retrieves the buffer's physical address and passes it to the Trusted OS through an SMC.
- The Trusted OS maps the physical memory of the buffer into the virtual address space of the requested TA.
- The TA processes the data in the shared buffer.

However, if the Trusted OS does not implement proper measures, there is a risk of double-fetch bugs. During the execution of the TA, the physical memory region containing the request may be accessible to the Normal World. This poses a potential security risk, as the buffer could be

concurrently modified by a Normal World thread running on another core. To mitigate this risk, the Trusted OS needs to implement synchronization mechanisms or validation checks to ensure the integrity of the shared buffer during its processing by the TA.

## 2.3 Static data-flow analysis

Static data-flow analysis employs various techniques to analyze how data flows through a program without executing it. Two primary approaches within static data-flow analysis are forward analysis and backward analysis.

Forward analysis involves tracing the flow of information in a program from its entry point to its exit points. In this approach, the analysis starts at the initial state of the program and progresses through its various components, tracing the propagation of information and dependencies as the program unfolds. Forward analysis is particularly effective for understanding how data is generated, modified, and utilized during the execution of a program. It enables the identification of potential issues, such as variable overwrites or undefined behavior, by examining how values flow through variables and expressions. Reaching definitions analysis is one of the applications of forward analysis, allowing us to determine which variable definitions may reach specific program points.

Conversely, backward analysis traces the flow of information from the exit points of a program back to its entry point. This retrospective approach provides insights into the impact of variables on the outcomes and results of the program. Backward analysis is valuable for understanding how certain variables contribute to specific outcomes, aiding in the identification of sources of errors or vulnerabilities. Live variable analysis is a notable application of backward analysis, where the focus is on identifying variables that remain in use until specific program points. This information is crucial for eliminating unused variables and optimizing the program for efficiency.

In addition to these general approaches, static data-flow analysis often involves the construction of Use-Def (Use-Definition) and Def-Use (Definition-Use) chains. These chains represent the relationships between the use of a variable and its definition within the program. Use-Def chains track where a variable is used and where it is defined, providing insights into the values it can take on during program execution. Def-Use chains, on the other hand, reveal where a variable is defined and where it is subsequently used. These chains play a crucial role in understanding the flow of data and dependencies within a program, aiding in the identification of potential issues such as variable misuse, uninitialized variables, or redundant computations.

## 2.4 Ghidra's P-Code

Ghidra is a distinguished open-source software reverse engineering tool developed by the National Security Agency (NSA), that allows to analyze binary executables. At the heart of Ghidra lies P-Code, an intermediate representation that exists at two levels of abstraction. The first is a direct one-to-many translation of the disassembled assembly instructions, called low or raw P-Code. The second is high or refined P-Code, which is the result of various transformations on the raw P-Code from Ghidra's decompiler. P-Code enables the seamless translation of assembly instructions from various CPU architectures into a common, abstracted form. This "lifting" process empowers reverse engineers by abstracting away the intricacies of diverse architectures, such as x86's multitude of instructions, MIPS' delay slots, and ARM's conditional instructions. The result is a unified and simplified instruction set that transcends architecture-specific complexities. P-Code lifting is characterized by a one-to-many translation, where a single assembly instruction may be represented by one or more P-Code instructions. P-Code works by translating individual processor instructions into a sequence of P-Code operations that take parts of the processor state as input and output variables (Varnodes). A Varnode is a generalization of either a register or a memory location, represented by the formal triple: an address space, an offset into the space, and a size. A P-Code operation is the analog of a machine instruction that takes one or more Varnodes as input and optionally produces a single output Varnode. The set of unique P-Code operations (distinguished by an opcode) comprises a fairly tight set of the arithmetic and logical actions performed by general-purpose processors. Its abstraction not only facilitates a standardized analysis workflow across disparate architectures but also allows reverse engineers to develop automated analyses more effectively. [1, 6]

# Chapter 3

# Problem

The Trusted Execution Environment (TEE) plays a pivotal role in securing sensitive operations and data within a processor. In ARM devices, TrustZone technology establishes a secure enclave, creating a separation between the Regular Execution Environment (REE) or *Normal World* (considered untrusted) and the Trusted Execution Environment (TEE) or *Secure World* (considered trusted).

A key concern arises in the communication between the Normal World and the Secure World, where Client Applications (CAs) interact with Trusted Applications (TAs) through a TEE driver and Secure Monitor Calls (SMCs). The critical issue lies in the handling of shared data buffers exchanged between these worlds. If not managed properly, there is a risk of double-fetch bugs, wherein the physical memory region containing the request may be accessible to the Normal World during the execution of a Trusted Application. This poses a significant security threat as the buffer could potentially be concurrently modified by a Normal World thread running on another core.

A real-world example of this vulnerability is highlighted by the work of Sanfelix [11], showcasing a double-fetch issue inherent to the shared memory interface in Samsung TEE. Samsung employs a TrustZone-based Trusted Execution Environment (TEE) in their smartphones for various sensitive tasks such as facilitating secure payments, managing cryptographic keys, and ensuring system integrity verification. Due to an incorrect use of the shared buffer, the main communication channel to send and receive data, arbitrary code execution was achieved. Specifically, the Client Application fills the shared memory and calls into the Trusted Application to do the processing. However, the Client Application can modify the data between the validation and the subsequent use, leading to a Time-of-Check-Time-of-Use bug.

The threat model involves the attack surface associated with the Trusted Execution Environment (TEE) on ARM devices. The attack vector involves exploiting double-fetch bugs in the communication process between the Normal World and the Secure World.
Specifically on Android devices, the real-world impact of the identified vulnerabilities in the TEE

poses a critical threat to the security and privacy of users. With many Android devices relying on ARM processors and incorporating TrustZone technology, the exploitation of double-fetch bugs in the TEE communication process could allow attackers to gain unauthorized access to sensitive data. This includes cryptographic keys and biometric information, leading to severe consequences such as unauthorized transactions, identity theft, or compromise of user privacy. Given such sensitive operations, device manufacturers and developers must implement robust security measures, timely software updates, and patches to mitigate the risk of exploitation and guarantee the reliability and trustworthiness of the Trusted Execution Environment and Trusted Applications.

# Chapter 4

# Design

In the design section, we detail the workflow of the static analyzer, the design choices, and the different steps involved in the identification of double-fetch bugs. We also provide a running example to better illustrate the process. In Figure 4.1, we present an overview of the static analysis process.

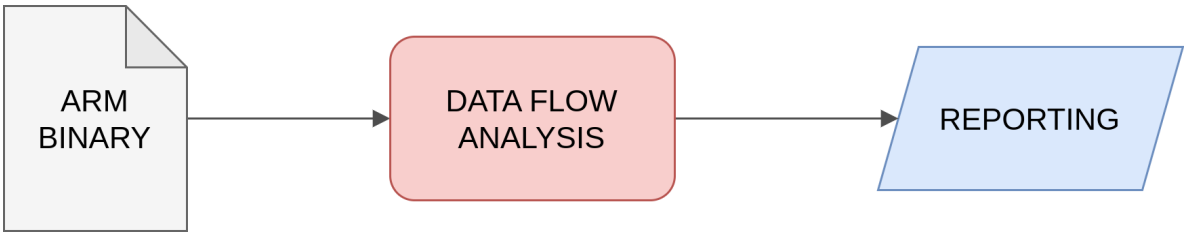


Figure 4.1: Static analysis workflow

We start with an ARM binary that we want to analyze, and then we run the data-flow analysis, which is the essence of our tool. There is a first initialization phase where the binary is decompiled and its function signatures are changed according to section 4.1, then we start the data-flow analysis from the entrypoint functions of the TAs. We process the descendants, we compute their types and we collect the fetch-use pairs. We check if they are connected to form a quadruple, as described in section 4.2. Finally, we render the results for debugging purposes and we report the connected fetch-use pairs to the user, for manual inspection and validation.

## 4.1 TAs Entrypoints

The Global Platform's TEE Internal Core API Specification [3] defines a framework for TAs as a set of Entrypoints called upon request by Normal World clients. There are three entrypoint functions, namely `TA_OpenSessionEntrypoint`, `TA_CloseSessionEntrypoint` and `TA_InvokeCommandEntrypoint`. As shown in Listing 4.1, `TA_OpenSessionEntrypoint` and `TA_CloseSessionEntrypoint` respectively opens and closes the session object, which is passed to the other functions. `TA_InvokeCommandEntrypoint` requests a command specified by the `cmd_id` parameter.

```
uint32_t TA_OpenSessionEntrypoint(void **session_obj,
          uint32_t param_types,
          TC_NS_Parameter params[4]);
uint32_t TA_InvokeCommandEntrypoint(void **session_obj,
          uint32_t cmd_id,
          uint32_t param_types,
          TC_NS_Parameter params[4]);
uint32_t TA_CloseSessionEntrypoint(void **session_obj);
```

Listing 4.1: Interfaces of TAs entrypoint functions

An array of four `params` is passed to `TA_OpenSessionEntrypoint` and `TA_InvokeCommandEntrypoint`. The `param_types` argument specifies the type of the four parameters, which can be input or output or both, as shown in Listing 4.2.

```
enum TEE_ParamType {
    TEE_PARAM_TYPE_NONE = 0x0,
    TEE_PARAM_TYPE_VALUE_INPUT = 0x1,
    TEE_PARAM_TYPE_VALUE_OUTPUT = 0x2,
    TEE_PARAM_TYPE_VALUE_INOUT = 0x3,
    TEE_PARAM_TYPE_MEMREF_INPUT = 0x5,
    TEE_PARAM_TYPE_MEMREF_OUTPUT = 0x6,
    TEE_PARAM_TYPE_MEMREF_INOUT = 0x7,
};
```

Listing 4.2: Parameter types in TEE

Each `params` is of type `TC_NS_Parameter`, which is a union of a `memref` structure, including the address and the size of a buffer, and a `value` structure, consisting of two int values, as can be seen in Listing 4.3.

```
typedef union {
    struct {
        void *buffer;
        uint32_t size;
```

```
    } memref;
    struct {
        int32_t a;
        int32_t b;
    } value;
} TC_NS_Parameter;
```

Listing 4.3: Parameter structure in TEE

Since the array of four `TC_NS_Parameter`, is accessible by the Normal World during a TA's execution, the `memref` structure is interesting in the context of race conditions. The Global Platform standard contains a warning regarding `memref` parameters: *"For a Memory Reference Parameter, the buffer may concurrently exist within the client and Trusted Application instance memory spaces. It SHALL therefore be assumed that the client is able to make changes to the content of this buffer asynchronously at any moment. It is a security risk to assume otherwise"* [3].

## 4.2 Double-fetch characterization

In order to start automatically detecting double-fetches, it is important to outline the concept of a *fetch-use pair*. A *fetch* is a read from one of the buffers, where a read-access memory dereference happens, a *use* is any operation performed on a fetched value. A fetch-use pair involves the same memory location and can be connected to another fetch-use pair to form a *quadruple*, leading to a potential double-fetch.

## 4.3 Fetch-use pairs identification

In this section, we are giving a general description of the core idea to identify the fetch-use pairs and then we are showcasing the different steps by examining a running example.
We first start from the `params` argument of the `TA_OpenSessionEntrypoint` and `TA_InvokeCommandEntrypoint` functions, and we recursively traverse all the descendants. We compute the type of each instruction and we collect the fetch-use pairs by tainting the fetch instructions and associating the subsequent use instructions.
When encountering a call to non-standard functions, we run an interprocedural analysis, by restarting the data-flow analysis procedure.
To compute all the *quadruples* (i.e., the connected fetch-use pairs) we leverage a control-flow analysis. We compute the basic blocks of each instruction and we consider two fetch-use pairs *FUP1* and *FUP2* to be connected if the basic block of the *FUP2* fetch instruction is reachable from the basic block of the *FUP1* use instruction.

We discuss next our running example (Listing 4.4) involving a typical double-fetch pattern in the `TA_InvokeCommandEntryPoint`, composed of two fetch-use pairs, with the first use being a conditional statement. Specifically, the `memref` buffer we are interested in is copied into the `path` variable at line 3. After that, the `path` is involved in a fetch with the `strlen` function at line 10. The resulting variable is then used in a conditional statement at line 11. The `path` is fetched again with another `strlen` function at line 13. The resulting variable is used as an argument to the `memcpy` function at line 14. This creates a *quadruple*.

```
1  uint32_t TA_InvokeCommandEntryPoint(void **session_obj, uint32_t cmd_id,
2                          uint32_t param_types, TC_NS_Parameter params[4]) {
3    char *path = (char *)params[3].memref.buffer;
4    int test_size0 = params[0].memref.size;
5    int test_size1 = params[1].memref.size;
6
7    char buffer[MAX_PATH_LEN + 1];
8    memset(buffer, 0, MAX_PATH_LEN + 1);
9
10   uint32_t path_len1 = strlen(path);
11   if (path_len1 + test_size1 < MAX_PATH_LEN + 1 + test_size0) {
12     memcpy(buffer, "ABCDEF", 6);
13     uint32_t path_len2 = strlen(path);
14     memcpy(buffer, path, path_len2);
15   }
16 }
```

Listing 4.4: Running example showcasing a double-fetch bug

Let us analyze and look closer into how the static analyzer would proceed to detect this double-fetch. As a first step, we initiate the analysis from the specific entrypoint function and we locate the symbols of the `params` in Listing 4.4 code at lines 3, 4, and 5. We only consider the `params` corresponding to a `memref` buffer in Listing 4.3, meaning we avoid the processing of `params` at lines 4 and 5. For each of the remaining ones, we start to process the descendants as shown in Algorithm 1. The descendants correspond to all the instances in the code where the `memref` buffer occurs (e.g., variable assignments, function calls). As can be seen in Figure 4.2, related to the `params` copied into `path` in the running example, the descendants are the nodes that can be reached by following edges in a data flow graph, providing insight into the propagation of information through a program.

If we focus on the `memref` buffer assigned to the `path` variable, the first-level descendants are at lines 10, 13, and 14 in Listing 4.4. Algorithm 1 first checks if the descendant has been visited. If not, it proceeds to:

- Create an instruction based on the current descendant.
- Compute the type of the instruction, as detailed in chapter 5. This step constructs the fetch-use

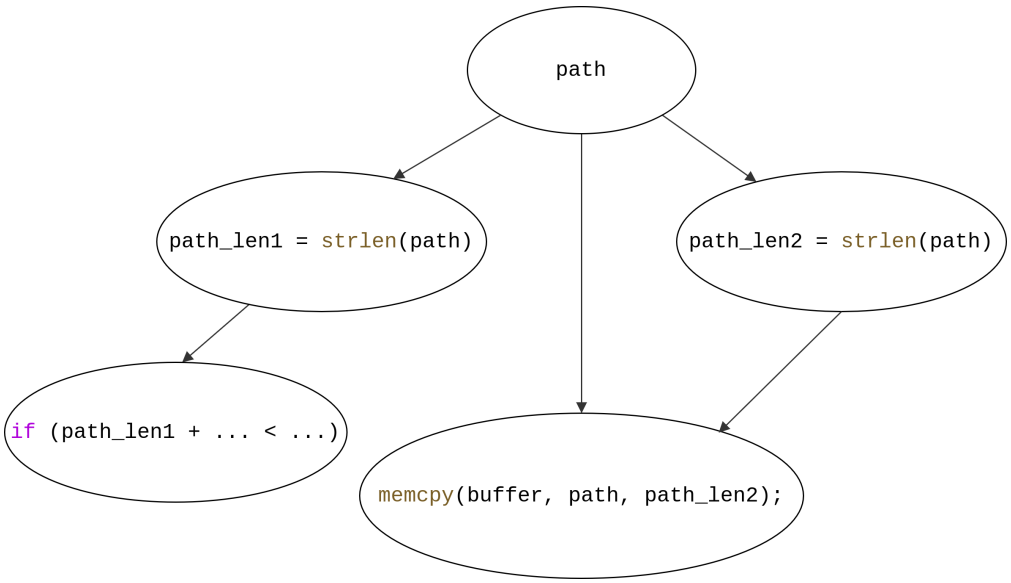


Figure 4.2: Running example's simplified data flow graph

**Algorithm 1** Process the descendants

```
1:  procedure ProcessDescendant(descendant, parent)
2:      if descendant is not already visited then
3:          ComputeInstructionType(instruction)
4:          add the instruction to the descendants list
5:          mark the descendant as visited
6:          add the descendant node to the data flow graph and connect it to the parent node
7:          output ← GetOutput(instruction)
8:          for all output_descendant in descendants_of_output do
9:              ProcessDescendant(output_descendant, descendant)
10:         end for
11:     end if
12: end procedure
```

pairs and is also responsible for interprocedural analysis.

- Add the instruction to the list of descendants.
- Mark the descendant as visited to prevent duplicate processing, when encountering loops or goto statements.
- Add the descendant node to the fetch-use graph and add an edge to connect it to the parent node.
- Get the output of the current descendant
- Recursively process the descendants of the output of the current descendant.

After executing Algorithm 1 on the running example code, the two fetch-use pairs described above will be successfully computed, as can be seen in Listing 4.5.

```
fetch1: uint32_t path_len1 = strlen(path)
use1: if (path_len1 + test_size1 < MAX_PATH_LEN + 1 + test_size0)

fetch2: uint32_t path_len2 = strlen(path)
use2: memcpy(buffer, path, path_len2)
```

Listing 4.5: The detected fetch-use pairs

Now it is time to check whether these two fetch-use pairs are actually connected, in order to construct a *quadruple*, forming a potential double-fetch. First, let us introduce the concept of *basic blocks* (i.e., code without any jump target), that constitute the nodes in a Control Flow Graph (CFG). This type of graph models the possible execution paths through a program with directed edges that represent jumps in the control flow. The Algorithm 2 traverses the fetch-use pairs list and checks which ones are connected, according to the definition in section 4.2. It does preliminary checks to avoid duplicates and ensure the second fetch and the first use are related to the same `params` index and memory area, and it calls the `are_bb_connected` function. This function, shown in Algorithm 3, takes as input the basic block of the first use and the second fetch, and recursively traverses the basic blocks associated with the destination addresses of the source basic block. It compares these addresses with the destination's basic block address.

For our running example, a simplified control flow graph is shown in Figure 4.3. We can see that the first fetch-use pairs belong to the Basic Block 0, which is connected to the Basic Block 1, containing the second fetch-use pair.
By traversing the CFG, the second block is reachable from the first one, thus these two fetch-use pairs are connected and they create a *quadruple*, which is added to the `connected_fetch_use_pairs` list. These can be reported to the user, who can manually inspect them to asses if they represent double-fetch bugs, and evaluate their severity and exploitability.

**Algorithm 2** Return the list of connected fetch-use pairs

```
1: function GETCONNECTEDFETCHUSEPAIRS
2:     connected_fetch_use_pairs ← []
3:     for all fetch_use1 in fetch_use_pairs do
4:         fetch1, use1 ← fetch_use1.fetch, fetch_use1.use
5:         use1_bb_index ← GETBBINDEX(use1)
6:         for all fetch_use2 in fetch_use_pairs do
7:             if (fetch_use1, fetch_use2) not already in connected_fetch_use_pairs then
8:                 fetch2 ← fetch_use2.fetch
9:                 fetch2_bb_index ← GETBBINDEX(fetch2)
10:                if use1 and fetch2 are related to the same params index and memory area and
   AREBBCONNECTED(use1_bb_index, fetch2_bb_index, []) then
11:                    connected_fetch_use_pairs.append((fetch_use1, fetch_use2))
12:                end if
13:            end if
14:        end for
15:    end for
16:    return connected_fetch_use_pairs
17: end function
```

**Algorithm 3** Check if two basic blocks are connected

```
1: function AREBBCONNECTED(bb_src, bb_dest, visited_addresses)
2:     if bb_src == bb_dest then
3:         return True
4:     end if
5:     if bb_src not found then
6:         return False
7:     end if
8:     for dest_addr in basic_blocks[bb_src].destination_addresses do
9:         if basic_blocks[bb_dst].start == dest_addr then
10:            return True
11:        else
12:            if dest_addr ∉ visited_addresses then
13:                visited_addresses.append(dest_addr)
14:                bb_new_src ← GETBBINDEX(dest_addr)
15:                if AREBBCONNECTED(bb_new_src, bb_dst, visited_addresses) then
16:                    return True
17:                end if
18:            end if
19:        end if
20:    end for
21:    return False
22: end function
```

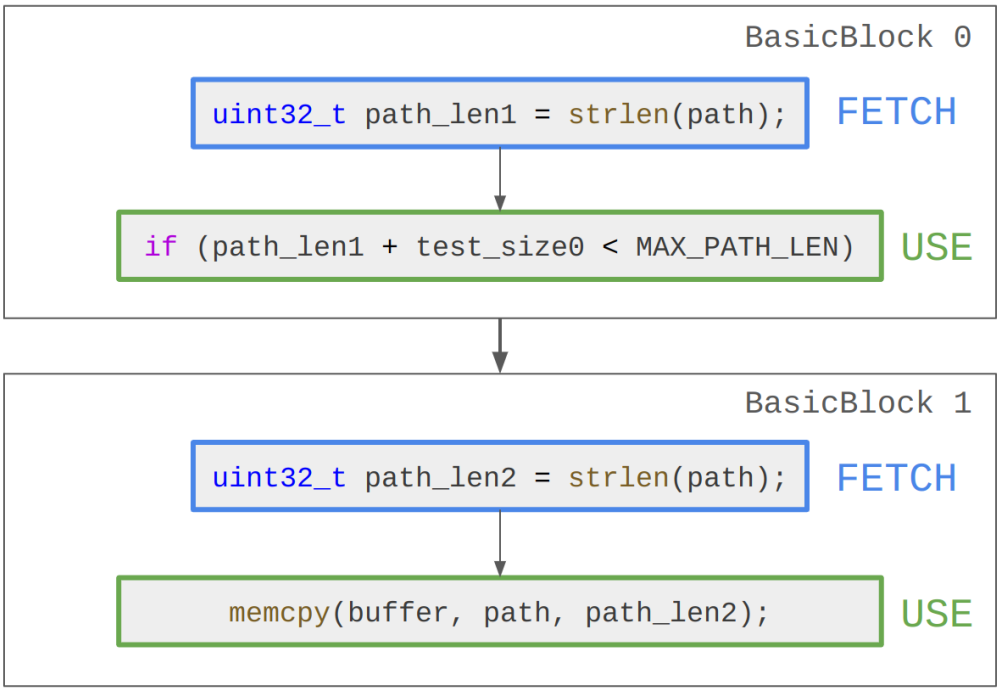


Figure 4.3: Running example's simplified control flow graph

When starting an interprocedural analysis the whole data-flow analysis is restarted and all the subsequent fetch-use pairs are collected into a global pool that starts from the original entrypoints (i.e., `TA_InvokeCommandEntryPoint` and `TA_OpenSessionEntryPoint`), to make sure that all the different quadruples can be connected across multiple functions.

## 4.4 Debugging and validation

For visualization and debugging purposes, the fetch and use instructions were plotted into a directed fetch-use graph, separated by the `params` index and the entrypoint function. Two nodes are connected if one is the descendant of the other. This allows us to have a nice overview of the relationships between the different instructions and validate their correctness.

Similarly, a report in JSON format was created to store all the connected fetch-use pairs for each TA, subdivided by `params` indexes. The aim is to store useful information to ease the manual analysis and validation of the double-fetches vulnerabilities.

# Chapter 5

# Implementation

This project is implemented in Python 3 and we chose Ghidra as the reverse engineering tool. Ghidra has a rich scripting API that perfectly fits our use case as we want to identify double-fetches in ARM binaries in an automatic way. Furthermore, Ghidra's P-Code intermediate representation is designed specifically to facilitate the construction of data flow graphs for follow-on analysis of disassembled instructions. An important property of Ghidra's P-Code is Static Single Assignment (SSA). This property requires each variable to be assigned exactly once and defined before it is used. This simplifies and improves the results of a variety of compiler optimizations and makes use-def chains explicit, each containing a single element.

To better understand the P-Code translation, let us consider an example where the decompiled instruction `__s = (char *)params[3].memref.buffer` is translated into raw and refined P-Code. In Listing 5.1, we can see that the refined P-Code is more compact, as it is composed of four instructions, and is tight to the decompilation. It has the explicit pointer addition of 3 to get the `params` index, the two pointer subtractions to access the first element of the `memref` buffer, and the final load to store into the local variable. On the other hand, as can be seen in Listing 5.2, the raw P-Code, while being more verbose and composed of more instructions, is more reliable, as there can be some decompilation mistakes, especially on the indexes when accessing some arrays. We can see the explicit addition of 24 to indicate we are considering the third `params` index since every params is composed of 8 bytes.

```
(register, 0x2c, 4) PTRADD (register, 0x2c, 4) , (const, 0x3, 4) , (const, 0x8, 4)
(unique, 0x10000092, 4) PTRSUB (register, 0x2c, 4) , (const, 0x0, 4)
(unique, 0x10000072, 4) PTRSUB (unique, 0x10000092, 4) , (const, 0x0, 4)
(unique, 0x10000096, 4) LOAD (const, 0x1a1, 4) , (unique, 0x10000072, 4)
```

Listing 5.1: Refined P-Code translation

```
$U2e80:4 = INT_RIGHT 24:4, 31:4
```

```
$U2f00:1 = INT_EQUAL 0:1, 0:1
$U2f80:1 = BOOL_AND $U2f00:1, CY
$U3000:1 = INT_NOTEQUAL 0:1, 0:1
$U3080:1 = SUBPIECE $U2e80:4, 0:4
$U3100:1 = BOOL_AND $U3000:1, $U3080:1
shift_carry = BOOL_OR $U2f80:1, $U3100:1
tmpCY = INT_CARRY r3, 24:4
tmpOV = INT_SCARRY r3, 24:4
r3 = INT_ADD r3, 24:4
tmpNG = INT_SLESS r3, 0:4
tmpZR = INT_EQUAL r3, 0:4

$U8280:4 = INT_ADD r3, 0:4
r3 = LOAD ram($U8280:4)
```

Listing 5.2: Raw P-Code translation

Ghidra supports a powerful scripting API that enables developers to programmatically analyze programs using Ghidra scripts, written in Java or Python 2.7 (via Jython). However, Python 2.7 reached the end of its life in 2020 and many tools developed by the security community are now written in Python 3.
Ghidrathon [4] is a Ghidra extension that adds Python 3 scripting capabilities to Ghidra. This allows us to use modern Python, including any third-party packages installed on the local system, to programmatically access the Ghidra database and the scripting's APIs. Moreover, Ghidrathon enables Python 3 scripts to be executed in headless mode to automate analysis across many files without the overhead of Ghidra's user interface.

This project is modular, incorporates error handling, and the code is documented to ensure ease of maintenance, code reusability, scalability, and collaboration. It is also dockerized to ensure portability and isolation. The run script takes as an argument a TA or a folder containing TAs and runs the main Ghidra script in headless mode.

The algorithm to identify double-fetches works as follows:

1. Initialize the program
2. Change the entrypoint functions signatures to update the P-Code translation
3. Decompile the functions
4. Start the data-flow analysis to process the `params` symbols and their descendants, compute the instruction types, and collect the fetch-use pairs
5. For each function and for each of the four `params`, output the connected fetch-use pairs into a JSON report and plot the fetch-use graph

We leveraged Ghidra's Program interface [7] that represents the main entry point into an object which stores all information relating to a single program, including the entrypoint functions that we are interested in.

We created a custom class `SignatureChanger` for managing the creation of new function signatures. By using Ghidra's `DataTypeManager` [8] we changed the `TA_OpenSessionEntrypoint` and `TA_InvokeCommandEntrypoint` signatures according to the interfaces in Listing 4.1. We also created a new data type representing the `params` structure `TC_NS_Parameter` following the Listing 4.3. We differentiated between 32-bit and 64-bit to specify the correct data type sizes, in order to match the underlying architecture. As a result, the refined P-Code gets translated accordingly to match the new signatures, allowing a better and more coherent analysis.

We created a `Decompiler` wrapper class around Ghidra's `DecompInterface` [9] to decompile the specified entrypoint functions.

We located all the symbols of the `params` inside the function using Ghidra's high-level abstractions and we collected their Varnode instances. By analyzing the underlying P-Code we considered only the `params` representing the buffer in the `memref` structure and we computed the corresponding index (from 0 to 3). We then got the first-level descendants and we started the data-flow analysis. For each descendant: we first checked if it has already been visited since there can be loops. If not, we calculated the descendant id and created an `Instruction` wrapper to ease the retrieval of information (e.g., address, output). We computed the instruction type, added the instruction to the descendants list, and marked it as visited. Finally, we calculated the output descendants and we restarted the data-flow analysis from each output in a depth-first search manner.

In the custom `Instruction` class, the output of an instruction is filtered based on the P-Code's opcode and the function name since opcode like `STORE` and functions like `memcpy` or `memmove` consider as output the second P-Code's input. This can be extended to support more opcodes and functions.

An instruction type is defined as: `NONE`, representing instructions where no memory dereferencing happens (e.g., adding, subtracting, casting), `FETCH` and `USE`, following the definitions in section 4.2. The instruction type is calculated via a custom `compute_instruction_type` function that analyses the raw and refined P-Code and outputs the corresponding type based on the opcode and contextual considerations. Initially, the function iterates over a list of tainted instructions, checking if any of their outputs match the inputs of the current instruction. If a match is found, the instruction is marked as already fetched, and the index of the fetch tainted instruction is recorded. The instruction type (i.e., `NONE`, `FETCH`, or `USE`) is then determined based on the opcode and the function name in case of `CALL`, `CALLIND` or `INDIRECT` opcodes. In fact, some standard functions, like `memcpy` or `strlen`, which involve memory dereferencing, don't need a separate data-flow analysis since their behavior is known beforehand and can be directly classified. On the other hand, an interprocedural analysis is run to categorize non-standard functions: the data-flow analysis is

restarted from the argument in the current instruction corresponding to the shared buffer, it keeps track of the `params` and collects the interprocedural fetch-use pairs so that they can be considered for connection.
Based on the opcode, an instruction is classified as a USE if it is involved in a store or in a comparison. If there are two consecutive LOAD instructions or a LOAD followed by an ADD, and then later another LOAD, it indicates a memory dereference has occurred. The dereference index is tracked to ensure the construction of fetch-use pairs considers instructions related to the same memory area. If the instruction is identified as a `FETCH` operation, it is appended to the list of tainted instructions, ensuring comprehensive tracking of data flow. If the instruction is marked as a `USE` operation and has already been fetched, a fetch-use pair is constructed and the instruction output is appended to the outputs of the corresponding tainted fetch instruction.

The fetch-use graph is constructed using the Python package NetworkX [5] and is outputted as a PDF file. Each instruction is added as a node in the graph and connected to its parent instruction. Each node is composed of its refined p-code, its address, and its type (if not NONE). Every analyzed TA generates a separate directed fetch-use graph for every function and for every `params` index, in the format `TAName_FunctionName_param_ParamIndex`.

A `BasicBlock` wrapper is created to represent a PCodeBasicBlock [10], with start, stop, and destination addresses. The basic blocks are initialized at the beginning of the data-flow analysis and sorted by start address. Each instruction belongs to a specific basic block with a `basic_block_index`, obtained by comparing the instruction address with each basic block start and stop address. After having collected all the fetch-use pairs, the quadruples are calculated by getting the connected fetch-use pairs. To check for connectivity, a custom `are_bb_connected` operates in a recursive manner as described in section 4.3.
A custom `ReportCreator` class initializes the JSON report and gradually adds the connected fetch-use pairs. For this purpose, a custom `FetchUsePair` wrapper class includes useful information to ease the subsequent manual analysis and keep track of the `params` index of each connected fetch-use pair.

# Chapter 6

# Challenges

Addressing Time-of-Check to Time-of-Use (TOCTOU) bugs in Trusted Execution Environments (TEEs) using our static approach comes with its own set of challenges:

1. **Limited Analysis Scope and complex Manual Analysis:** Performing a large-scale analysis across multiple Trusted Applications (TAs) is inherently challenging. This approach requires access to the corresponding devices and specific firmware versions, and the collection of a large set of associated Trusted Applications binaries. Furthermore, the subsequent manual analysis is challenging and time-consuming, due to the size and complexity of real-world TAs. Thus, we aimed to ease the analyst's job of evaluating the double-fetch bugs, by including useful information in the reports to allow the identification of the root cause. For future work, a more sophisticated tracking of the buffer modifications could be implemented, by including a list of reference calls and traces of the instructions that interact with the involved parameters.

2. **Shortcomings of Binary Analysis:** Static analysis, while feasible, relies on binary analysis and thus faces its inherent limitations. The lack of access to source code and the obfuscated nature of binaries hinder a complete understanding of the code's logic and potential vulnerabilities, making it challenging to precisely identify and evaluate security issues, and potentially leaving exploits undetected. Some contextual information, such as variable names, comments, and high-level constructs, is not available. Furthermore, dynamic behaviors, such as runtime interactions, input-dependent branching, and memory allocation, are challenging to infer accurately without the actual execution of the program. We are leveraging Ghidra's capabilities to analyze the binaries. Thus, we rely on the fact that Ghidra is continuously been updated alongside its binary analysis and reverse engineering functionalities. This will inherently allow us to conduct a more thorough static analysis.

3. **False Positives in Static Analysis:** Static analysis, while a valuable tool, is susceptible to generating false positives. The complexity of TEE implementations, and proprietary Trusted

Applications (TAs), may lead to misinterpretations of potential vulnerabilities. Distinguishing between intentional security measures and actual risks poses a significant challenge, introducing the possibility of inaccurate threat assessments. It is necessary to extensively test the static analyzer to identify more edge cases and integrate new functionalities to address them effectively.

4. **Inconsistencies in P-code Translation:** The translation of Ghidra P-code may not always align perfectly with the decompiled code. Therefore, it is necessary to continuously expand the static analyzer to cover as many cases as possible and cope with inconsistent P-code translation. Particularly, it is crucial to keep debugging and refine the implementation to address particularly challenging cases, such as when dealing with many indexes, multiple fetches and uses in the same instruction, or complex functions with a lot of calls and pointer manipulations.

# Chapter 7

# Evaluation

## 7.1 Test Suite

To showcase some relevant double-fetch patterns, a test suite has been created. The Trusted Applications have been developed on the open-source OP-TEE Trusted OS [13] and each one implements a different double-fetch pattern. The TAs are compiled and the static analyzer tool is executed on the ELF binary files. The results of the analyses are reported in the following sections.

### 7.1.1 Simple intraprocedural test case

In this TA we are simulating a simple double fetch vulnerability corresponding to an intraprocedural fetch-use fetch-use pattern. As seen in Listing 7.1, the `memref` buffer is copied to a local buffer and it is fetched by calling the `strlen` function on it. After that, there are two uses of the `memref` buffer since the `len` variable appears in the if statement and as an argument to the `malloc` function. The same situation happens a second time with the `buf2` and `len2` variables. This intraprocedural fetch-use fetch-use pattern is successfully detected by the static analyzer tool. Four quadruples are reported since there are 2 fetches and 2 uses.

```
1 TEE_Result TA_InvokeCommandEntryPoint(..., TEE_Param params[4]) {
2     ...
3     char *buf = (char *)params[0].memref.buffer;
4     uint32_t len = strlen(buf);
5     if (len > 0) {
6         char *test_buf = malloc(len);
7     }
8
9     char *buf2 = (char *)params[0].memref.buffer;
```

```
10      uint32_t len2 = strlen(buf2);
11      if (len2 > 0) {
12          char *test_buf = malloc(len2);
13      }
14      ...
15  }
```

Listing 7.1: Simple intraprocedural test case

### 7.1.2 Nested conditional statements intraprocedural test case

This TA aims to test the static analyzer to find connected fetch-use pairs when many basic blocks are involved. As seen in Listing 7.2, the `memref` buffer is copied into a local buffer and a fetch happens to calculate its length with the `strlen` function. The `len` variable is then used in a series of if statements and in a call to `malloc`. To bypass the first four checks the `len` variable must be greater than 0, less than 10, even, and divisible by 3. Thus it must be equal to 6. However, after these checks, the buffer's length is fetched again with the `strlen` function and then it is used in the `malloc`. Since different checks are performed, the previous constraints are no longer valid and the length could assume now any even value less than 10, leading to a TOCTOU situation.
In this code snippet, there are seven basic blocks, two fetches (i.e., the two calls to `strlen`), five uses related to the first fetch (i.e., at lines 6, 7, 8, 9, and 12) and three uses related to the second one (i.e., at lines 15, 16 and 17). The static analyzer reports 15 connected fetch-use pairs, successfully considering all the basic block connections and the combinations of fetches and uses.

```
1   TEE_Result TA_InvokeCommandEntryPoint(..., TEE_Param params[4]) {
2       ...
3       char *buf = (char *)params[0].memref.buffer;
4       uint32_t len = strlen(buf);
5
6       if (len > 0) {
7           if (len < 10) {
8               if (len % 2 == 0) {
9                   char *test_buf = malloc(len);
10                  char *buf2 = (char *)params[0].memref.buffer;
11
12                  if (len % 3 == 0) {
13                      uint32_t len2 = strlen(buf2);
14
15                      if (len2 < 10) {
16                          if (len2 % 2 == 0) {
17                              char *test_buf = malloc(len2);
18                          }
19                      }
```

```
20                 }
21             }
22         }
23     }
24     ...
25 }
```

Listing 7.2: Nested if statements test case

### 7.1.3 Indexing test cases

The purpose of this test case is to test whether the static analyzer can correctly distinguish indexes when the `memref` buffer is dereferenced, ensuring that the connected fetch-use pairs are related to the same memory area. We can see from Listing 7.3 that in both the `TA_InvokeCommandEntryPoint` and `TA_OpenSessionEntryPoint`, the `memref` buffer is dereferenced and its element is copied into two local variables, constituting a fetch operation. These are used as the size parameter of the `malloc` functions to create two different buffers. However, while in the `TA_InvokeCommandEntryPoint` the `memref` buffer is dereferenced on the same index 0, in the `TA_OpenSessionEntryPoint` we are first getting the element at index 0 and then the element at index 2. Thus, the static analyzer correctly detects only the quadruple in the `TA_InvokeCommandEntryPoint` because the two fetches in the `TA_OpenSessionEntryPoint` functions are related to different memory areas.

```
1  TEE_Result TA_InvokeCommandEntryPoint(..., TEE_Param params[4]) {
2      ...
3      uint32_t a = ((uint32_t *)params[0].memref.buffer)[0];
4      void *buf_a = malloc(a);
5
6      uint32_t b = ((uint32_t *)params[0].memref.buffer)[0];
7      void *buf_b = malloc(b);
8      ...
9  }
10
11 TEE_Result TA_OpenSessionEntryPoint(..., TEE_Param params[4], ...) {
12     ...
13     uint32_t a = ((uint32_t *)params[0].memref.buffer)[0];
14     void *buf_a = malloc(a);
15
16     uint32_t b = ((uint32_t *)params[0].memref.buffer)[2];
17     void *buf_b = malloc(b);
18     ...
19 }
```

Listing 7.3: Memory dereference test case

As in the previous test case, in this TA we are testing whether the static analyzer correctly handles indexing situations. As seen in Listing 7.4, here we are copying the first element of the `memref` buffer to two local variables, but we are accessing `params` at two different indexes (i.e., 0 and 2). The static analyzer correctly doesn't detect any quadruples since the two fetch-use pairs are related to different parameters.

```
1  TEE_Result TA_InvokeCommandEntryPoint(..., TEE_Param params[4]) {
2      ...
3      uint32_t a = ((uint32_t *)params[0].memref.buffer)[0];
4      void *buf_a = malloc(a);
5
6      uint32_t b = ((uint32_t *)params[2].memref.buffer)[0];
7      void *buf_b = malloc(b);
8      ...
9  }
```

Listing 7.4: Different params indexes test case

### 7.1.4 Simple interprocedural test case

This TA aims to show the interprocedural analysis capabilities of our static analyzer. As seen in Listing 7.5, in the `TA_InvokeCommandEntryPoint` function the `memref` buffer is assigned to `src` and its size is calculated with the `strlen` function. Then `src` is used as the source and fetched in the `memcpy` call. The length of the resulting `dst` buffer is calculated with the `strlen` function, representing a use. After this, there is a call to the `FUN_buffer_alloc` function, where the original buffer `src` is passed as the first argument. The function first calculates the length of the buffer using the `strlen` function(i.e., a fetch) and then allocates memory for a new buffer of the same length using the `malloc` function(i.e., a use). The original `memref` buffer is fetched twice in two different functions. This represents an interprocedural double-fetch pattern, that is correctly detected by our static analyzer.

```
1  static TEE_Result FUN_buffer_alloc(void *buffer) {
2      uint32_t len = strlen((char *)buffer); // fetch
3      char *a = malloc(len);              // use
4
5      return TEE_SUCCESS;
6  }
7
8  TEE_Result TA_InvokeCommandEntryPoint(..., TEE_Param params[4]) {
9      ...
10     char *src = (char *)params[0].memref.buffer;
11     uint32_t src_size = params[0].memref.size;
12     char *dst = malloc(src_size);
```

```
13
14      memcpy(dst, src, src_size); // fetch
15      uint32_t len = strlen(dst); // use
16
17      FUN_buffer_alloc(src);
18      ...
19  }
```

Listing 7.5: Simple interprocedural test case

## 7.2 Real-world TA analysis

In this section, we evaluate our static analyzer tool to test its effectiveness in detecting bugs in some real-world TAs.

### 7.2.1 Huawei P9 Lite

#### The `task_storage` TA

As a first real-world TA, we analyzed the `task_storage` TA of a Huawei P9 Lite (VNS-L31) with firmware version C432B160. This TA encrypts any data written to a file and decrypts it when reading from one, implementing an encrypted file system. It uses Persistent Objects, that are part of Global Platform's TEE specification [3].
The `TA_InvokeCommandEntrypoint` function consists of a huge switch-case statement over the `cmd_id`. As shown in Table 7.1, the `cmd_id` specifies the command to execute and it is associated with a type. We are only interested in the `MEMREF` type, which represents the shared buffer, which could be fetched twice.

As seen in Listing 7.6, an interesting first bug occurs in the `FOpen` command. It expects the first parameter to be a `MEMREF_INPUT` containing a path in the form of a null-terminated string residing in a shared buffer. The path length is calculated using `strlen` and it is then checked to be below a maximum value. The result of a second `strlen` call is used to copy the path to a heap buffer. This poses a classical TOCTOU race condition. Even if the first check succeeds, the path could be extended before the second `strlen` and `strncat`. If by that time the string is long enough, the heap buffer will be overflown, leading to a heap-based buffer overflow and a controllable modification of heap control structures. Apart from crashing the TA, this concurrency bug could be used in a heap exploitation scenario.

```
1  #define MAX_PATH_LEN 128
2  uint32_t package_name_len = *((uint32_t *) session_obj);
```

| cmd_id | Name | Parameter 0 | Parameter 1 | Parameter 2 | Parameter 3 |
|---|---|---|---|---|---|
| 0x11 | FOpen | MEMREF_INPUT | VALUE_INPUT | VALUE_OUTPUT | - |
| 0x12 | FClose | VALUE_INPUT | - | - | - |
| 0x13 | FClear | - | - | - | - |
| 0x14 | FRead | VALUE_INPUT | MEMREF_OUTPUT | VALUE_OUTPUT | - |
| 0x15 | FWrite | VALUE_INPUT | MEMREF_INPUT | VALUE_OUTPUT | - |
| 0x16 | FSeek | VALUE_INPUT | VALUE_INPUT | - | - |
| 0x19 | FClose_Delete | VALUE_INPUT | - | - | - |
| 0x1a | FInfo | VALUE_INPUT | VALUE_OUTPUT | - | - |
| 0x1b | FSync | VALUE_INPUT | - | - | - |

Table 7.1: Commands in the `task_storage` TA and their expected parameter types

```
3  char *package_name = (char *) session_obj + 4;
4  char *buffer = TEE_Malloc(MAX_PATH_LEN + package_name_len + 1);
5  /* path is a pointer to a shared buffer */
6  uint32_t path_len1 = strlen(path);
7  if (path_len1 + 17 < MAX_PATH_LEN + 1) {
8      memcpy(buffer, "/sec_storage_data", 17);
9      uint32_t path_len2 = strlen(path);
10     strncat(buffer, path, path_len2 + 1);
11 }
12 TEE_Free(buffer);
```

Listing 7.6: Simplified code excerpt from task_storage FOpen command

As a first step to start automatically detecting double-fetches using static analysis, we developed a POC code to showcase the concurrency bug above. This allowed us to isolate the problem, validate our reasoning, and improve our solution incrementally. The C code was compiled in a 64-bit machine using arm gcc to ensure the resulting assembly code is tailored for the ARM architecture and follows ARM calling conventions.

We then ran the Python script to start the static analysis and looked at the generated report. We successfully discovered the double-fetch bug, identified as `TS-FOPEN-RC`, in both the POC code and

the TA's code.

Regarding the `TA_OpenSessionEntrypoint` function, it requires the fourth parameter to be a `MEMREF_INPUT` containing the client's package name, residing in shared memory. Using `TEE_MemCompare`, it verifies if the package name matches either *"/system/bin/tee_test_store"* or *"com.huawei.hidisk"*. If the match is successful, a heap buffer is allocated as the session context. Then, the package name length is calculated using the `strlen` function and it is stored in the first four bytes of the buffer, followed by the package name itself. The double-fetch bugs are related to the package name length fetched twice, and to the check of the package name against the above strings and the subsequent copy to the heap buffer.
Additionally, it's worth noting that the `task_storage` TA doesn't involve interprocedural calls (i.e., everything happens inside the `TA_InvokeCommandEntrypoint` and `TA_OpenSessionEntrypoint` functions), making the review process simpler.
After manually reviewing the generated report, we discovered 3 quadruples related to the double-fetch bug in the `TA_InvokeCommandEntrypoint` function, along with 46 false positives, and 2 quadruples related to the double-fetch bugs in the `TA_OpenSessionEntrypoint` function, along with 7 false positives. Since we found all the bugs we were aware of, we didn't report any false negatives.

#### **The** `task_keymaster` **TA**

This TA allows Client Applications to securely use cryptographic keys, by keeping them secret to a potential attacker. As can be seen in Table 7.2, the `TA_InvokeCommandEntryPoint` calls one of those functions, based on the `cmd_id`.

This TA contains two race conditions in the `generate_keyblob` function, which is called by the `km_generate_key` and `km_import_key` functions. The first race condition, identified as `KM-GENERATE-KEYBLOB-RC1`, is related to the initial `key_characteristics`, which are copied from the input buffer to the generated keyblob. However, at least two of the `key_characteristics` parameters (i.e., algorithm and key size) were already fetched. If they have been altered, they could potentially create an inconsistent keyblob that does not match the key material. There is another race condition in the same function, identified as `KM-GENERATE-KEYBLOB-RC2`. The function doesn't use a private heap or stack buffer while creating the new keyblob. Rather, it stores the keyblob data in steps using the output buffer and then expects it to stay that way. The HMAC is calculated and saved in the first 0x20 bytes of the output buffer at the end of the function. The keymaster signs a keyblob that contains data it did not create directly if a Normal World thread overwrites keyblob members before the HMAC is produced. If additional functions receive the poisoned blob and believe it to be authentic because it has the correct HMAC, then this becomes critical.
Due to the complexity of these situations, which involve many function calls and require an understanding of the relationships between keyblobs and key materials, the static analyzer could not find

| cmd_id | Name |
|---|---|
| 0x1 | km_generate_keypair |
| 0x2 | km_get_keypair_public |
| 0x3 | km_import_keypair |
| 0x4 | km_sign_data |
| 0x5 | km_verify_data |
| 0x6 | km_init_ability |
| 0x7 | km_generate_key |
| 0x8 | km_get_key_characteristics |
| 0x9 | km_import_key |
| 0xa | km_export_key |
| 0xb | km_begin |
| 0xc | km_update |
| 0xd | km_finish |
| 0xe | km_abort |

Table 7.2: Commands offered by the task_keymaster TA

these two bugs.

The function km_begin is used to initialize a cryptographic operation. An example is the decryption operation in Listing 7.7. There is a race condition, identified as KM-HMAC-VERFICATION-RC, regarding the shared buffer used in the HMAC verification. The keyblob, which is still in shared memory, is expected to remain the same if the operation is successful. Concurrent modifications, however, could make any data read from the shared buffer before or during the verification invalid. There are several dereferences of the keyblob pointer. The length of the encrypted parameters and any additional data they include are first obtained. Next, off_enc_params is appended twice to the keyblob's base address to obtain a pointer to the start of enc_params. If off_enc_params have been altered, the resulting address would point to an incorrect location. The same bug exists in the km_export_key function. However, due to the limitations of the static analyzer when handling multiple indexes, this bug could not be found.

```
1  uint32_t km_begin(int32_t paramTypes, TC_NS_Parameter params[4]) {
2    char local_hmac[0x20];
3    keyblob_t *keyblob = params[0].memref.buffer;
4    uint32_t len_keymaterial = keyblob->len_keymaterial;
5    /* ... */
6    keymaster_HMAC((char *) keyblob + 0x20, params[0].memref.size-0x20,
7    local_hmac);
8    if (TEE_MemCompare(local_hmac, keyblob, 0x20) == 0) {
9      ...
10     /* decrypt the keyblob */
11     uint32_t len_enc_params = keyblob->len_enc_param_data
12     + keyblob->off_enc_param_data - keyblob->off_enc_params;
13     void *tmp_buf = TEE_Malloc(len_enc_params);
14     TEE_MemMove(tmp_buf, keyblob + keyblob->off_enc_params, len_enc_params);
15     keyblob_crypto(keyblob->iv, tmp_buf, len_enc_params,
16     (char *)keyblob + keyblob->off_enc_params, 1);
17   }
18 }
```

Listing 7.7: Decryption operation in km_begin

The km_get_key_characteristics function contains the race conditions above, as it performs the same HMAC verification and parameter decryption operations. After that, the key characteristics are copied from the keyblob, which resides in shared memory, to the output buffer, as can be seen in Listing 7.8. The keyblob is dereferenced multiple times and its off_hw_params, off_param_data, and len_param_data members determine what data to copy. However, if those values were modified, there could be a leak of sensitive data. Once again, due to the indexing limitations when handling many dereference operations, this bug could not be found.

```
1  uint32_t len_characteristics = keyblob->off_param_data
2      + keyblob->len_param_data - keyblob->off_hw_params;
```

```
3  TEE_MemMove(params[2].memref.buffer,
4      (char *) keyblob + keyblob->off_hw_params,
5      len_characteristics);
```

Listing 7.8: Excerpt from km_get_key_characteristics

The static analysis resulted in a total of 15 false positives and 8 false negatives, as the above bugs were not found.

#### The `task_gatekeeper` TA

This TA is used for authenticating authenticating user passwords and generating authentication tokens. The `TA_InvokeCommandEntrypoint` function calls two functions: `enroll` and `verify`. When `enroll` is invoked without the first two parameters, it calls the `create_gatekeeper_handle` function, which can be seen in Listing 7.9. This function initializes parts of the `gatekeeper_handle`, that corresponds to the shared output buffer, and passes it to the `password_handle_hmac` function together with the password blob. This function copies parts of the `gatekeeper_handle` and the password blob into a temporary heap buffer, and stores the computed HMAC in `dest`, which points to the `gatekeeper_handle`'s hmac.
The first race condition, identified as `GK-CREATE-HANDLE-RC`, is related to the `gatekeeper_handle` construction, that uses the shared output buffer. Before `password_handle_hmac` uses the data added to the handle in `create_gatekeeper_handle` to build the HMAC, the data might have been overwritten. Additionally, after `create_gatekeeper_handle` returns, the function `add_fail_record` is invoked and the id argument is fetched from the shared buffer again, which creates a potential race condition.
When enroll is invoked and the first two parameters are also input buffers, `compare_and_update` is called. Another potential race condition, identified as `GK-UPDATE-HANDLE-RC`, could occur. While the gatekeeper handle is in the shared buffer, the TA verifies it. If successful, the relevant `fail_record` is reset by fetching the handle's id. However, if the id was concurrently changed after the verification, the `fail_record` of an entry could be reset, even though there was no corresponding authentication. To create the updated gatekeeper handle, the id is also fetched again. Consequently, the modified `gatekeeper_handle` may have a different id than the original one. The `verify` function is also vulnerable to this race condition.

```
1  uint32_t create_gatekeeper_handle(gatekeeper_handle_t *handle, uint32_t unused,
2    uint8_t *password_blob, uint32_t pw_blob_size) {
3      handle->version = 2;
4      TEE_GenerateRandom(&handle->id_l, 8);
5      handle->unknown3 = 0;
6      handle->unknown4 = 0;
7      TEE_GenerateRandom(handle->iv,8);
8      handle->flag63 = 1;
```

```
9     password_handle_hmac(handle, password_blob, pw_blob_size, handle->hmac);
10    /* ... */
11 }
12
13 uint32_t password_handle_hmac(gatekeeper_handle_t *handle, uint8_t *pw_blob,
14   uint32_t pw_blob_size, uint8_t *dest) {
15    uint8_t *tmp = TEE_Malloc(0x11 + pw_blob_size);
16    *tmp = handle->version;
17    memcpy(tmp + 1, &handle->id_l, 8);
18    memcpy(tmp + 9, &handle->unknown3, 8);
19    TEE_MemMove(tmp + 0x11, pw_blob, pw_blob_size);
20    gatekeeper_HMAC(tmp, pw_blob_size + 0x11, handle->iv, dest);
21    /* ... */
22 }
```

Listing 7.9: Simplified code excerpt in create_gatekeeper_handle and password_handle_hmac

All three bugs were not found by the static analyzer, and it reported four false positives, because of its strict behavior that tends to separate multiple different indexes related to the same instruction.

### 7.2.2 Redmi 9A

#### Keymaster TA

This TA represents the Keymaster of a Redmi 9A, stored in the binary `d78d338b1ac349e09f65f4efe179739d.ta`. It has access to the raw key material, and it is responsible for managing cryptographic keys and providing secure key storage and operations.
The static analyzer successfully found a double-fetch in the function `FUN_0000a594` on `params[1].memref.buffer`. This function is reachable from `TA_InvokeCommandEntrypoint` through a series of interprocedural calls. As can be seen in Listing 7.10, the double-fetch situation is similar to the one in section 7.2.1. `param_2`, which originated from `params[1].memref.buffer`, is fetched in the first `strlen` call, used in a conditional statement, fetched again, and the resulting size `sVar2` is used as the `size` parameter of a `memcpy` call. `param_3`, which is the result of this final `memcpy`, points to a stack buffer of fixed size. Thus, it could be overflown if `sVar2` is sufficiently large.

```
1 // param_2 was originally params[1].memref.buffer
2 ...
3 sVar2 = strlen(param_2); // FETCH1 on param_2
4 if (*param_4 < sVar1 + sVar2 + 1) { // USE1 on sVar2
5    uVar3 = 1;
6 }
7 else {
8    memset(param_3,0,*param_4);
```

```
9     sVar1 = strlen(param_1);
10    memcpy(param_3,param_1,sVar1);
11    sVar1 = strlen(param_1);
12    sVar2 = strlen(param_2); // FETCH2 on param_2
13    memcpy((void *)((int)param_3 + sVar1),param_2,sVar2); // USE2 on sVar2
14    ...
```

Listing 7.10: Code excerpt from FUN_0000a594 highlighting the fetches and the uses

The static analyzer successfully detected 3 connected fetch-use pairs related to the bug above, along with 15 false positives, due to inaccuracies when dealing with memory function parameters.

### 7.2.3 Samsung Galaxy S6

#### The `TEE_Keymaster` TA

This TA is part of the Samsung Galaxy S6 Edge TEE with firmware version G925FXXU6ERC4. It represents the keymaster, and it is stored in the binary `07060000000000000000000000000000.tlbin`. Based on the `cmd_id`, the shared TCI buffer is passed to one of the functions in Table 7.3.

As can be seen in Listing 7.11, the `RSAGenerateKeyPair` function initially sanitizes and copies some values from the TCI into local variables. The requested `key_pair_type` is then used to initialize a `tlApi_key_pair`. Next, `tlApiGenerateKeyPair` receives as input the `tlApi_key_pair` and fills the buffers with randomly generated key material. Finally, by repeatedly copying the generated components into a local buffer, the actual `keyblob` is created.

```
1  uint32_t RSAGenerateKeyPair(tci) {
2      tlApi_key_pair_t tlApi_key_pair;
3      uint32_t key_size_bytes;
4      keyblob_t keyblob;
5
6      uint32_t key_size_bits = tci->key_size_bits;
7      if (key_size_bits <= 0x1000) {
8          key_size_bytes = (key_size_bits+7) / 8;
9          if (tci->key_pair_type == RSA)
10             init_RSA_key_pair(&tlApi_key_pair, key_size_bytes);
11         else
12             init_RSA_CRT_key_pair(&tlApi_key_pair, key_size_bytes);
13         tlApiGenerateKeyPair(&tlApi_key_pair);
14         keyblob.key_size_bits = tci->key_size_bits;
15         if (tci->key_pair_type == RSA)
16         make_keyblob_from_RSA_key_pair(&tlApi_key_pair, &keyblob);
17         else
18         make_keyblob_from_RSA_CRT_key_pair(&tlApi_key_pair, &keyblob);
```

| cmd_id | Name |
|---|---|
| 0x1 | RSAGenerateKeyPair |
| 0x2 | RSASign |
| 0x3 | RSAVerify |
| 0x4 | KeyImport |
| 0x5 | GetPubKey |
| 0x6 | DSAGenerateKeyPair |
| 0x7 | DSASign |
| 0x8 | DSAVerify |
| 0x9 | ECDSAGenerateKeyPair |
| 0xa | ECDSASign |
| 0xb | ECDSAVerify |
| 0xc | GetKeyInfo |

Table 7.3: Commands offered by the TEE_Keymaster TA

```
19          /* wrap keyblob in a Secure Object and return it to the Normal World */
20      }
21  }
```

Listing 7.11: The RSAGenerateKeyPair function

A first race condition, identified as TEE-KM-GENERATE-RC1, occurs because the `key_size_bits` is fetched from the TCI twice. First, it is stored in a local variable and then, after sanitization, it is used for computing the corresponding byte size. However, the value is fetched from the TCI again to set the keyblob's `key_size_bits`. This poses a TOCTOU since the second value could be larger than 0x1000, bypassing the conditional statement constraint.
A second race condition, identified as TEE-KM-GENERATE-RC2, occurs because the `key_pair_type` is fetched from the TCI twice. Once to determine the initialization of `tlApi_key_pair`, and then to decide how the `tlApi_key_pair` should be interpreted for the keyblob creation. If `key_pair_type` was concurrently modified, there could be a type-confusion when `tlApi_key_pair` is evaluated.

The function `RSASign` expects the TCI to contain addresses and sizes of three additional buffers, containing a wrapped keyblob, the data to be signed, and an output buffer for the signature. The addresses and sizes are first stored in local variables, and the memory ranges are validated. Next, as can be seen in Listing 7.12, the wrapped keyblob is copied into a stack buffer and verified using the tlApiUnwrapObject function. tlApi functions are used to generate the signature, and they expect a `tlApi_key_pair_t` structure as a key representation. A race condition, identified as `TEE-KM-SIGN-RC`, starts from the `tlApi_key_pair`, which is initialized using the members of the local keyblob copy. However, for setting the `ptr_priv_exp`, a value from the shared buffer at offset 36 corresponding to the length of the plain data is fetched. This value could have been changed since the wrapped keyblob was copied to a local buffer and verified. As a result, `ptr_priv_exp` could point to a faulty location.

```
1  uint32_t RSASign(tci) {
2      /* fetch and sanitize addresses in tci */
3      tlApi_key_pair_t tlApi_key_pair = {0};
4      wrapped_keyblob_t local_wrapped_keyblob;
5      memcpy(&local_wrapped_keyblob, shared_wrapped_keyblob, shared_len);
6      tlApiUnwrap(&local_wrapped_keyblob)
7      /* initialize tlApi_key_pair using local_wrapped_keyblob ... */
8      /* initialize pointer to private key */
9      tlApi_key_pair.ptr_priv_exp = &local_wrapped_keyblob
10     + SIZE_SECURE_OBJECT_HEADER
11     + shared_wrapped_keyblob->len_plain_data;
12     tlApiSign(tlApi_key_pair, data, out_buf);
13 }
```

Listing 7.12: Code excerpt from the RSASign function

The static analyzer detected 83 false positives and did not find the above `TEE_Keymaster`'s race conditions, because the P-Code translation was not reliable, as it was based on the raw binary. To cope with this, we developed some POC code to isolate and showcase the bugs above. Those were successfully detected by the static analyzer. As Trustonic's Kinibi TEE uses the proprietary MobiCore Load Format (MCLF), for future work, a Ghidra's MCLF loader could be used to check the effectiveness of our tool and see whether we can obtain more significant results.

#### The `HDCP` TA

The *High-bandwith Digital Content Protection* (HDCP) TA is used for encrypting media data. It is stored in the binary `ffffffff000000000000000000000005.tlbin`. In the function handler of the command `0xe2`, there is a race condition, identified as `HDCP-RC`. As seen in Listing 7.13, the TA fetches a value from the shared buffer and uses it as the size parameter for `tlApiMalloc`. Then, the same value is fetched again and used to set a local heap buffer to zero. Finally, it is fetched for the third time when used as the `memcpy` size and source parameter.

```
1 uint32_t command_0xe2_handler(uint32_t param_1, uint32_t *buf) {
2     /* ... */
3     uint8_t *heap_buf = tlApiMalloc(shared_buf[0x4002]);
4     if (heap_buf != NULL) {
5         set_zero(heap_buf, shared_buf[0x4002])
6         memcpy(heap_buf, shared_buf, shared_buf[0x4002]);
7         /* ... */
8     }
9 }
```

Listing 7.13: Code excerpt from the command_0xe2_handler function

This bug was successfully detected by the static analyzer in 13 different quadruples. However, the false positives rate is quite high (i.e., 218), as this TA is large and complex, so many false positives originate from memory dereferences.

### 7.2.4 Evaluation results

In the Table 7.4 we can see an overview of the different false positive (FP), true positive (TP), and false negative (FN) rates related to the number of connected fetch-use pairs for each double-fetch bug in every analyzed TA. From the high number of false negatives and false positives of some TAs, we can see evidence of the discussed limitations of the static analyzer. These include situations where a lot of indexes are involved or with functions containing a lot of calls and pointer manipulations, resulting in multiple fetches and uses in the same instruction.

| Trusted Application (TA) | Counts | | |
|---|---|---|---|
| | **False Positives (FP)** | **True Positives (TP)** | **False Negatives (FN)** |
| `p9lite_task_storage` | 53 | 5 | 0 |
| `p9lite_task_keymaster` | 15 | 0 | 8 |
| `p9lite_task_gatekeeper` | 4 | 0 | 3 |
| `redmi9a_keymaster` | 15 | 3 | 0 |
| `galaxys6_tee_keymaster` | 83 | 0 | 3 |
| `galaxys6_hdcp` | 218 | 13 | 0 |

Table 7.4: FP, TP, FN rates of the evaluated TAs

Analyzing many large Trusted Applications manually takes a lot of time. In order to obtain meaningful results, we would need to conduct a large-scale static analysis experiment to analyze numerous TAs. In this report, we didn't have enough time to gather more accurate statistics on false positives and false negatives. The size and complexity of TAs make it challenging to do a thorough manual analysis within our time constraints. Despite this limitation, our patterns and analyses provide a good starting point for understanding and addressing double-fetch vulnerabilities in Trusted Applications using static analysis.

# Chapter 8

# Related Work

## 8.1 Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels

DEADLINE is a static analysis tool that can automatically detect double-fetch bugs in operating system kernels. The paper makes the following contributions:

1. They propose a formal and precise definition of double-fetch bugs that eliminates the need to manually verify whether a multi-read is a double-fetch bug.
2. They present the design and implementation of DEADLINE, an end-to-end system to automatically vet kernel code with a tailored symbolic execution model specifically designed for double-fetch bug detection.
3. They find and report 23 new bugs in the Linux kernel and a new bug in the FreeBSD kernel.
4. They propose four generic strategies to patch and prevent double-fetch bugs based on their study and the discussion with kernel maintainers.

Regarding contribution 1., they define a Fetch as a pair ($A$, $S$) where $A$ represents the starting address of the fetch and $S$ represents the size of the memory (in bytes) copied into the kernel. DEADLINE labels an execution path as a double-fetch bug when the following four conditions are met:

1. There are at least two reads from user-space memory, thus, it must be a multi-read.
2. The two fetches must cover an overlapped memory region in the user-space. Given two fetches, ($A_0$, $S_0$) and ($A_1$, $S_1$), they are considered to have an overlapped region if and only if: $A_0 <= A_1 < A_0 + S_0$ or $A_1 <= A_0 < A_1 + S_1$. They use a pair ($A_{01}$, $S_{01}$) to denote the overlapped

memory region for the two fetches and a triple ($A_{01}$, $S_{01}$, $i = [0,1]$) to denote the memory copied in during the first or second fetch.

3. A relation must exist based on the overlapped regions between the two fetches. A control dependence relation consists of a variable $V \in (A_{01}, S_{01}, 0)$ subject to a set of constraints in order for the second fetch to happen. A data dependence relation consists of a variable $V \in (A_{01}, S_{01}, 0)$, that is consumed, such as being assigned to other variables, involved in calculations, or passed to function calls.

4. DEADLINE cannot prove that the relation established still holds after the second fetch, leading to a race condition capable of destroying this relation.

Regarding contribution 2., the DEADLINE double-fetch detection procedure is composed of two steps:

1. FINDING MULTI-READS: Scan the kernel to collect as many multi-reads as possible and construct their execution paths.

2. FROM MULTI-READS TO DOUBLE-FETCH BUGS: Using symbolic checking along each execution path to determine if a multi-read turns into a double-fetch bug.

Regarding step 1., DEADLINE first compiles the kernel source code into the LLVM intermediate representation (IR) and statically analyzes the IR to identify multi-reads and prune associated execution paths. Initially, DEADLINE identifies all fetches in the kernel, and, starting at each fetch, within the function it resides in, DEADLINE scans through both the reaching and reachable instructions for this fetch, and among those instructions, either marks that a fetch pair has been found or inlines the function containing a fetch and re-executes the search. In addition to the two fetches F0 and F1, the enclosing function $F_n$ is also attached to the pair, denoting a multi-read as the triple ($F_0$, $F_1$, $F_n$). This allows DEADLINE to find all execution paths within the enclosing function $F_n$ with a CFG traversal and to slice out the irrelevant instructions that have no impact on the fetches or are not affected by them. The last step is to linearize the paths into a sequence of IR instructions. If a path does not have loops, DEADLINE concatenates the basic blocks, while for a path with loops, DEADLINE performs unrolling only once.

Regarding step 2., DEADLINE first transforms the LLVM IR to a symbolic representation SR, symbolically executing the LLVM instructions along the path and adding the constraints to the assertion set for the SMT solver. All the SRs are derived from a set of root SRs, which could be function arguments, global variables, or KMEM and UMEM, that represent memory blobs in kernel and user-space. To symbolically execute branch instructions, DEADLINE looks ahead on the path, checks which branch is taken, and uses this information to infer the constraints that must be satisfied by taking that branch. Also, common functionalities such as memory allocations (`kmalloc`), memory operations (`memcpy`), string operations (`strlen`), synchronization operations (`mutex_lock`), debug

and error reporting functions(`printk`) are abstracted out by manually writing symbolic rules that capture the interactions between the function arguments and the return values symbolically.

In the traditional symbolic executors model memory, two reads from the same address are assumed to always return the same value if there is no store operation to that address between the reads. This does not hold for multi-threaded programs with user-space accesses from kernel code. Thus, DEADLINE extends the memory model by encoding a monotonically increasing epoch number in the reads from user-space memory to represent that the values copied in at different fetches can be different. In addition, instead of assuming the whole memory to be an array of bytes, DEADLINE represents each memory object by an array of bytes and maps each pointer to one memory object based on the following rules:

1. Different function arguments or global variables are assumed to be pointing to different memory objects.
2. Newly allocated pointers are assumed to be pointing to new memory objects.
3. When there is an assignment, the pointed object is transferred.
4. For any other pointer, if it is impossible to prove that its value falls in the range of any existing object, it is assumed that it points to a new object.

Finally, DEADLINE invokes the SMT solver. Specifically, DEADLINE first checks whether the two fetches share an overlapped memory region, and then it asks the solver to check whether there exist any overlapped regions with all the assertions. Finally, for each overlap, DEADLINE checks whether there is control dependence or data dependence established based on this region. If even all the checks succeed, the multi-read is marked as a double-fetch bug, otherwise, it is considered safe.

## 8.2 RacerX: Effective, Static Detection of Race Conditions and Deadlocks

RacerX is a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. It found serious errors when applied to Linux, FreeBSD, and a large commercial code base.

RacerX is composed of five phases:

1. Retargeting it to system-specific locking functions. (manual)
2. Extracting a control flow graph from the checked system.
3. Running the deadlock and race checkers over this flow graph.

4. Post-processing and ranking the results.

5. Inspection. (manual)

In the context of double-fetches, this tool requires a final manual inspection to check if a multi-read is a double-fetch.

To retarget it to a new system, the user supplies a table specifying functions used to acquire and release locks as well as those that disable and enable interrupts. In addition, users may optionally provide an annotator routine that marks whether routines are single-threaded, multi-threaded, or interrupt handlers.

The extraction phase iterates over each file in the checked system and extracts a control flow graph file containing all function calls, uses of global variables, uses of parameter pointer variables, and, optionally, uses of all local variables. To support flow-sensitive analysis, each statement has a set of pointers to all statements that immediately follow it in the program text.

The analysis phase reads the emitted CFG files for the entire system into memory, constructs a linked whole-system CFG, and then traverses it checking for deadlocks or races. Given the set of roots (i.e., functions without callers), the analysis phase iterates over each and does a flow-sensitive, depth-first, interprocedural traversal of the CFG, tracking the set of locks held at any point. At each program statement, the race detection or deadlock checkers receive the current statement, the current lockset, and other information, which they use to emit error messages and statistics used during the final inspection phase.

The final phase consumes the analysis results and post-processes them to compute ranking information for error messages based on the likelihood of being a false positive, and the difficulty of inspection. It then presents these to the user for manual inspection.

Regarding the lockset analysis to detect both deadlocks and race conditions, RacerX computes locksets using a top-down, flow- and context-sensitive, interprocedural analysis. It starts from the root of each call graph and does a depth-first search (DFS) traversal down the control flow graph (CFG), analyzing the effects of each path, and the lockset at each actual callsite. Conceptually, the DFS traversal adds and removes locks as needed and calls the race and deadlock checkers on each statement in the flow graph, caching analysis results at both the statement and function level.

## 8.3 RELAY: Static Race Detection on Millions of Lines of Code

RELAY is a static and scalable algorithm that can perform race detection on large programs.

RELAY uses a technique called relative lockset analysis to address the limitations of static race detection. Relative lockset analysis describes the changes in the locks being held relative to the

function entry point. These relative locksets allow RELAY to summarize the behavior of a function independently of the calling context.

RELAY uses a bottom-up context-sensitive analysis over the call graph. This analysis starts from the entry point of the program and then follows the call graph to analyze each function. For each function, RELAY first uses relative lockset analysis to compute the lockset for each instruction. Then, RELAY uses the lockset information to check for races.

For each function being analyzed, RELAY performs these steps:

1. Symbolic execution: RELAY performs symbolic execution to track the values of variables and the locks that are held at each point in the program. This information is used to compute the relative locksets.

2. Relative lockset analysis: RELAY computes the relative locksets for each function. The relative lockset for a function is a disjoint pair of locksets representing the locks that are definitely acquired on all executions of the function, and the ones that may have been released on some executions of the function.

3. Guarded access analysis: RELAY computes the guarded accesses for each function. A guarded access is a triple *(lvalue, lockset, access kind)*, where *lvalue* is the memory location that is being accessed, *lockset* is the relative lockset at the point of the access, and *access kind* is either read or write.

4. Race detection: RELAY compares pairs of guarded accesses whose *lvalues* may be aliases, whose positive locksets have an empty intersection, and where at least one of the accesses is a write. If such a pair of accesses is found, then RELAY reports a race warning.

# Chapter 9

# Conclusion

In our study, we presented a static analysis approach to automatically detect double-fetches in Trusted Applications. Initially, we underscored the critical challenge of concurrency issues leading to Time-Of-Check to Time-Of-Use (TOCTOU) situations in Trusted Execution Environments (TEEs). We implemented a static analyzer in Python using Ghidra's scripting capabilities and its P-Code abstractions. We tested our tool on different Trusted Applications and successfully identified various double-fetch patterns. For future work, the plan is to test the tool on a bigger set of Trusted Applications and expand the set of supported functions and P-Code operations, in order to increase the accuracy and discover more bugs. Static analysis could be combined with symbolic execution or concolic execution to find the inputs that trigger the involved fetches, simplifying the manual analysis. Finally, evaluating the security impact of a double-fetch is a critical task that could be automated as much as possible, in order to reduce the operational effort of a potential security analyst. A fuzzer could be used, by overcoming the challenge of virtualizing TEEs, or the reporting could be further enhanced, by precisely tracking the buffer modifications in an automatic way.

# Bibliography


[1] Alexei Bulazel. *Working with Ghidra's P-Code to identify vulnerable function calls*. 2019. URL: https://riverloopsecurity.com/blog/2019/05/pcode.

[2] Dawson Engler and Ken Ashcraft. *RacerX: effective, static detection of race conditions and deadlocks*. ACM SIGOPS Operating Systems Review. 2003.

[3] 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_%20TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf.

[4] Mike Hunhoff. *Ghidrathon: Snaking Ghidra with Python 3 Scripting*. 2022. URL: https://github.com/mandiant/Ghidrathon.

[5] NetworkX developers. *NetworkX - NetworkX Documentation*. URL: https://networkx.org.

[6] National Security Agency (NSA). *A Brief Introduction to P-Code*. 2020. URL: https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html.

[7] National Security Agency (NSA). *Interface Program*. URL: https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/Program.html.

[8] National Security Agency (NSA). *Interface DataTypeManager*. URL: https://ghidra.re/ghidra_docs/api/ghidra/program/model/data/DataTypeManager.html.

[9] National Security Agency (NSA). *Class DecompInterface*. URL: https://ghidra.re/ghidra_docs/api/ghidra/app/decompiler/DecompInterface.html.

[10] National Security Agency (NSA). *Class PcodeBlockBasic*. URL: https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/PcodeBlockBasic.html.

[11] Eloi Sanfelix. *Tee exploitation*. 2019. URL: https://labs.bluefrostsecurity.de/files/TEE.pdf.

[12] Michael Schwarz, Daniel Gruss, Clémentine Maurice Moritz Lipp, Thomas Schuster, Anders Fogh, and Stefan Mangard. *Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features*. pages 587–600, 05 2018. doi: 10.1145/3196494.3196508. 2017.

[13] TrustedFirmware.org. *OP-TEE Documentation*. 2023. URL: https://optee.readthedocs.io/.

[14] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. *RELAY: Static Race Detection on Millions of Lines of Code*. Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2007.

[15] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. *Precise and scalable detection of double-fetch bugs in os kernels*. IEEE International Symposium on Security and Privacy. 2018.