Master's Thesis
(Course Code: XM_0123)

Friddle: An Instruction-Level Dynamic Taint Analysis Framework for Detecting Data Leaks on Android and iOS

by

Simon Gao
(Student Number: 2761882)

Submitted in partial fulfillment of the requirements
for the degree of
'Computer Security' Master of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

August 26, 2025

August, 2025, Vrije Universiteit Amsterdam

Certified by .....
Herbert Bos
Full Professor
First Supervisor

Certified by .....
Floris Gorter
PhD Student
Daily Supervisor

Certified by .....
Erik van der Kouwe
Assistant Professor
Second Reader

Friddle: An Instruction-Level Dynamic Taint Analysis Framework for Detecting Data Leaks on Android and iOS

Simon Gao
Vrije Universiteit Amsterdam
Amsterdam, NL
sga207@student.vu.nl

ABSTRACT

Mobile applications often use native code to process sensitive data, but existing dynamic taint analysis tools are not enough in this area. For Android, most tools focus on the Java runtime, and tools for native code are rare. For iOS, such tools are even more limited. This creates a clear technical gap in tracking information flow at the native code level.

To address this problem, we designed and built Friddle, a dynamic taint analysis framework. Based on Frida's Stalker module, our framework operates at the instruction level. It first rewrites the native instruction stream at a compile time stage to inject analysis code. This code is then executed at run time to track data flows. We use bitmaps and interval trees to precisely manage taint states in registers and memory. The framework supports both Android and iOS and requires no modifications to the application or the operating system.

We evaluated Friddle using FriddleBench, our custom benchmark suite for testing taint analysis tools on mobile native code. Our results indicate that the framework successfully tracks data propagation in complex scenarios, including direct data transfers, implicit flows from table lookup operations, and data processed by standard encryption algorithms.

Our work makes three main contributions: a modular, cross platform dynamic taint analysis engine for native code on both Android and iOS; experimental validation of its tracking capabilities in realistic scenarios; and FriddleBench as an extensible evaluation platform for future research.

1 INTRODUCTION

As of 2025, Android and iOS hold 74.26% and 25.39% of the global mobile operating system market share [32]. With the steady growth of mobile applications, users face increasing risks of personal data leakage. Many applications request access to sensitive user information, such as SMS messages, call logs, geographic location, and contacts. However, users often do not know how this data is processed at runtime or whether it is sent to external servers [11]. Recent data protection regulations, such as GDPR and CCPA, require effective monitoring of data flows in mobile applications. Studies indicate that non compliance remains a major issue [16, 28].

Information flow tracking, especially taint analysis, is the main technical method for identifying such data leaks. This technique can be divided into static and dynamic methods based on when the analysis is performed.

Static Taint Analysis (STA) inspects program code without executing it. For example, `FlowDroid` can cover all possible execution paths to find potential data leaks by analyzing the source code or bytecode of Android applications [5]. However, static analysis methods struggle to handle correctly behaviors that can only be determined at runtime, such as Java reflection, dynamic code loading, and interactions with native code. These limitations can lead to imprecise analysis results when dealing with modern, complex applications [25, 35].

Dynamic Taint Analysis (DTA) monitors data flow while the program is running [30]. Systems like `TaintDroid` implement real time data flow tracking by modifying the Android operating system framework [11]. Because dynamic analysis only observes paths that are actually executed, it can handle correctly the dynamic behaviors that static analysis cannot, leading to more correct results. However, the effectiveness of dynamic analysis depends on the execution paths triggered by input test cases. If a test fails to cover a specific code branch, potential leaks in that branch will not be detected. Furthermore, fine grained dynamic monitoring often causes major performance overhead [2, 33].

Although existing dynamic taint analysis tools have made important progress on the Android platform, most of them focus on the Java runtime environment, the Android Runtime (ART). However, research shows that many applications implement core or sensitive logic in native code to improve performance or reuse existing codebases [1]. Currently, there are few dynamic taint analysis tools for Android native code, and existing solutions have limitations in terms of usability and coverage [7]. This creates a major analysis gap.

At the same time, on the iOS platform, due to its closed system, there is less public, fine grained taint analysis research. Existing work is mostly limited to high level function hooking, making it difficult to go deep to the instruction level to track data flow within functions.

To fill the gap in dynamic taint analysis for Android native code and the iOS platform, this research designed and built a framework called Friddle. Friddle is a cross platform, instruction level dynamic taint analysis engine based on dynamic binary instrumentation. It can perform fine grained information flow tracking for native code on mainstream Android and iOS devices without modifying the operating system or application source code.

1.1 Research Questions

To address the identified gaps in mobile native code taint analysis, this research investigates the following questions:

RQ1: Correctness In scenarios with different data transformations (e.g., basic data operations, table lookups and encryption), can Friddle correctly detect whether tainted data has moved from its source to its sink? This includes both true positive detection and false positive avoidance.

RQ2: Performance Overhead How much performance overhead does Friddle's dynamic instruction level instrumentation add to a target application compared to native execution?
RQ3: Instruction Coverage What proportion of the ARM64 instruction set, particularly data flow related instructions, can Friddle support?

The main contributions of this paper are as follows:

- We designed and built a modular and extensible dynamic taint analysis engine.
- We showed the engine's ability to track information flow in native code on both major mobile platforms, Android and iOS.
- We tested the effectiveness of Friddle in various scenarios, including implicit flows and cryptographic operations, using FriddleBench.

Friddle's core architecture is based on dynamic binary instrumentation. It uses Frida's Stalker engine to capture and rewrite native basic blocks at runtime. During this process, our propagation engine analyzes each ARM64 instruction and injects the corresponding analysis logic. This logic is then triggered as the program executes, enabling real time data flow monitoring. To handle complex instruction semantics accurately, we designed specific propagation rules for the ARM64 instruction set that address register aliasing, SIMD operations, and implicit flows from address calculations.

We evaluated Friddle using FriddleBench. Results indicate that the framework correctly tracks data propagation in complex scenarios, including implicit flows from table lookup operations and encrypted data processing. We tested both true and false positive detection, confirming that Friddle correctly distinguishes tainted from clean data. Our ARM64 instruction coverage analysis of 635 Android system libraries indicates that Friddle supports 98.5% of actual instruction execution frequency, focusing on the most critical data flow patterns.

The remainder of this paper is organized as follows. §2 introduces the relevant background knowledge. §3 details the design and implementation of Friddle. §4 describes our evaluation methodology and analyzes the experimental results. §5 discusses the limitations of our work and future directions. §6 reviews related work in the field. Finally, §7 concludes the paper.

2 BACKGROUND

2.1 Mobile Operating System Overview

Android and iOS are the two dominant mobile operating systems in the current market. Although they differ greatly in system architecture, application development languages, and ecosystem, both rely heavily on native code execution. An understanding of their respective native layer environments is basic for performing cross platform, low level analysis.

2.2 Android System Architecture

Android is an open source operating system based on the Linux kernel. It provides two core execution environments for applications:

Java Environment. This is the primary runtime for Android applications. Developers typically write application logic using Java or Kotlin. This code is compiled into Dalvik Executable (DEX) bytecode and executed within a specialized virtual machine called ART [20]. ART is responsible for application lifecycle management, automatic memory collection, and converting DEX bytecode into machine instructions.

Native Environment. The Android platform allows applications to load and execute native code written in languages such as C/C++ through the Java Native Interface (JNI). To support this, Google provides the Android Native Development Kit (NDK), a toolset that allows developers to implement parts of their applications using native code [18]. This code is compiled into platform specific shared libraries (`.so` files) that can be executed directly by the processor, free from the resource management constraints of the ART virtual machine. Performance intensive tasks are often implemented in the native layer. These include game engines, graphics rendering, audio/video decoding, and cryptographic algorithms. Data can flow bidirectionally between the Java managed environment and the native environment through JNI.

2.3 iOS System Architecture

iOS is a closed source operating system developed by Apple for its mobile devices. It is based on the XNU kernel and employs a strict, layered architecture to ensure system stability and security. Application developers mainly use Swift or Objective-C. The functionality of applications written in these high level languages heavily depends on underlying system frameworks, such as those in the Core Services and Core OS layers. These frameworks exist as pre compiled dynamic libraries (`.dylib` files).

All code written in Swift or Objective-C is converted by a compiler into native machine code for execution on the processor. Notably, the dynamic method invocation mechanism of Objective-C is implemented through a core function named `objc_msgSend`. This function acts as a central hub for message dispatch, serving as the bridge between method calls on objects and their underlying native implementations.

2.4 AArch64 Instruction Set Architecture

AArch64 is the 64-bit ARM instruction set architecture (ISA) [3] commonly used on modern mobile processors, providing a uniform execution foundation for all native code on both Android and iOS. It defines the program's execution model, including register usage and function calling conventions.

2.4.1 Register Set and Function. The AArch64 architecture defines a complete set of registers.

- **General Purpose Registers:** There are 31 64-bit general purpose registers, denoted `X0` through `X30`, used for data manipulation and address calculations. They can also be accessed as 32-bit registers (`W0` through `W30`).
- **Special Purpose Registers:**
  - `X29` serves as the Frame Pointer (FP), pointing to the base of the current function's stack frame for stable access to local variables.
  - `X30` is the Link Register (LR), which automatically stores the return address when a function is called via the `BL` (Branch with Link) instruction.
  - The `SP` is the Stack Pointer, always pointing to the top of the stack.

– A Zero Register (`XZR` or `WZR`), whose value is always zero, is also available.

2.4.2 Procedure Call Standard. AArch64 adheres to the "Procedure Call Standard for the ARM 64-bit Architecture" (AAPCS64). This standard dictates how functions pass and receive arguments and return values.

- **Argument Passing:** The first eight integer or pointer arguments to a function are passed via registers `X0` through `X7`.
- **Return Values:** A function's return value is typically placed in register `X0`.

2.4.3 Advanced SIMD and Floating Point. AArch64 integrates the NEON engine, which supports Single Instruction, Multiple Data (SIMD) operations. It includes 32 128-bit vector registers (`V0` through `V31`), enabling efficient parallel processing of data. This is crucial for accelerating tasks such as graphics, audio, and cryptographic computations.

These vector registers can be accessed in different sizes to accommodate both scalar and vector operations:

- **Scalar Operations:** Individual floating point values can be accessed as `S0-S31` (32-bit single precision) or `D0-D31` (64-bit double precision).
- **Vector Operations:** Full 128-bit SIMD operations use `Q0-Q31` registers, allowing parallel processing of multiple data elements in a single instruction.

2.5 Dynamic Binary Instrumentation (DBI)

DBI is a technique for analyzing and modifying the behavior of a program at runtime, without requiring access to its source code. It operates by injecting analysis code at the native instruction level, enabling the monitoring of a program's execution flow, memory access, and function calls.

Prominent DBI frameworks include Intel Pin [21], primarily used for the x86 architecture; DynamoRIO [10], a cross platform framework; and Frida [13], a lightweight toolkit that offers good support for both Android and iOS platforms.

2.5.1 The Frida Framework. Frida uses a client/server architecture. The `frida-server` component runs with high privileges on the target device (e.g., a rooted Android device or a jailbroken iOS device). An external control script, typically written in a language like Python on a PC, uses Frida's APIs to direct the server to inject a JavaScript based agent into the target process. This Frida Agent is the core component that executes all low level operations, responsible for implementing instruction level instrumentation, memory access, and communication with the control script.

To support the development of analysis logic, the Frida Agent also hosts a JavaScript engine (such as Google's V8 or the more lightweight QuickJS) within the process. This engine provides developers with a high level, easy-to-use JavaScript API, allowing them to control the Agent's underlying analysis and modification tasks by writing scripts. The core APIs provided by the Agent are organized into several modules, including:

- **Interceptor:** Used for implementing function level hooking.
- **Memory:** Provides a set of interfaces for directly reading and writing the target process's memory.
- **Process:** Used to query runtime information about the target process.
- **Stalker:** Frida's code tracing engine [15], which is the core for implementing instruction level analysis.

Stalker's Implementation Principles. Frida's Stalker is a Just-In-Time (JIT) compilation based code tracing engine. Its core logic is implemented in C within Frida's Gum library [14], and it provides the foundation for fine grained, instruction level analysis. Its working principle is as follows: when `Stalker.follow()` is called to trace a thread, Stalker takes control of that thread's execution flow. Whenever the thread is about to execute a new native basic block, Stalker first dynamically recompiles that block into a new memory region (a code cache). During this recompilation process, Stalker invokes a developer provided `transform` callback function.

Within the `transform` callback, the analysis script can iterate through every machine instruction in the basic block. Using the `iterator.putCallout()` method, a call to a predefined JavaScript function (the callout) can be inserted before or after any instruction. After instrumentation, Stalker executes the modified version of the code located in the code cache. This cycle of "copy basic block -> instrument and modify -> execute" repeats continuously as the program runs.

2.6 Jailbreaking and Rooting

Modern mobile operating systems, both Android and iOS, enforce a strict App Sandbox mechanism. Under this mechanism, each application runs in an isolated and restricted environment, with its access to the file system, hardware, and other processes being tightly controlled. While the sandbox mechanism greatly improves overall system security, it also creates a major obstacle for security analysis tools that require cross application or system level monitoring.

To bypass sandbox restrictions for in depth analysis, researchers usually need to gain privileged access to the device.

- On the iOS platform, the process is called **Jailbreaking**. It exploits software or hardware vulnerabilities—such as the checkm8 hardware exploit [34] utilized by tools like checkra1n [9] and palera1n [24]—to remove the software execution restrictions imposed by Apple. This allows users to access the entire operating system file system and enables analysis tools to interact with any process on the system.
- On the Android platform, the related process is known as **Rooting**. Similar to jailbreaking, rooting can be achieved by exploiting vulnerabilities in the operating system or kernel to gain privileged access. However, due to the open nature of the Android Open Source Project (AOSP) [19], a more common approach for developers is available on certain devices. This involves unlocking the device's bootloader (e.g., on a Google Pixel) and flashing a custom system image, such as a self compiled AOSP version that grants root permissions. This process allows a user or tool to obtain the highest level of system permissions (superuser) and thereby bypass standard sandbox limitations.

These elevated privileges are necessary for dynamic analysis tools to bypass sandbox restrictions and access system level resources required for instruction level monitoring.

2.7 Taint Analysis

Taint analysis is a program analysis technique used to track information flow. The core idea is to first mark sensitive data that needs protection (such as user passwords or geographical locations) as "tainted," then monitor the propagation path of this tainted data through the program, and finally examine if it flows to an external exit without authorization. This process involves three core concepts:

- **Taint Sources:** These are the points where sensitive data is generated. For example, a function that reads the user's contacts, an API call that retrieves GPS location information, or a memory buffer holding user input can all be defined as taint sources.
- **Taint Sinks:** These are the exit points where tainted data could be leaked. Typical sinks include network sending functions, file writing functions, or logging APIs.
- **Taint Propagation:** This describes the rules by which tainted data is passed from one variable to another or copied from one memory location to another during program execution. For instance, in the assignment operation `y = x`, if `x` is tainted data, then `y` should also be marked as tainted.

Based on when the analysis is performed, taint analysis is mainly divided into two methods: static and dynamic.

2.7.1 Static Taint Analysis. Static taint analysis involves analyzing the source code or binary code of a program without actually running it. It builds models such as the program's Control Flow Graph (CFG) and Data Flow Graph (DFG) to infer whether tainted data can reach a taint sink from a taint source across all possible execution paths.

Advantages Its main advantage is the ability to theoretically cover all of the program's execution paths, which helps in discovering potential data leak paths that might not be triggered in a specific set of test cases.

Disadvantages The main challenge for STA lies in its precision. It often struggles to accurately handle complex runtime behaviors such as reflection calls, dynamic code loading, and indirect jumps, which can lead to a large number of false positives.

2.7.2 Dynamic Taint Analysis. Dynamic taint analysis monitors the flow of data in registers and memory in real time while the program is running. It marks data as it is produced by a taint source and then updates and propagates the taint status according to each instruction that is executed.

Advantages DTA can precisely track real data flows because it only analyzes the code paths that are actually executed. This allows it to effectively handle dynamic behaviors that are difficult for static analysis, resulting in a very low false positive rate.

Disadvantages Its main drawback is that the scope of analysis is heavily dependent on the test cases provided. If a test case fails to trigger a specific code path, any potential data leak within that path will not be discovered. Additionally, because it requires monitoring at the instruction level, DTA typically introduces significant performance overhead.

Although the theory and techniques of dynamic taint analysis are relatively mature [30], there is currently a lack of a public, easy-to-use, and dedicated dynamic taint analysis framework for mobile platforms, especially for Android native code and the entire iOS platform. This research work aims to fill this gap.

2.7.3 Implicit Information Flows. While explicit information flows involve direct data movement (e.g., `y = x`), implicit flows occur when sensitive information affects program behavior indirectly. There are two main categories of implicit flows that present challenges for taint analysis systems.

Address-based Implicit Flows. These occur when tainted data influences memory access addresses, commonly seen in table lookup operations:

```
// Tainted index used for table lookup
char tainted_index = get_user_input();
char result = lookup_table[tainted_index]; // Implicit flow
```

Such patterns are common in encoding algorithms (e.g., Base64) and cryptographic operations (e.g., AES S-Box lookups), where input data serves as an index to select values from predefined tables.

Control-flow Implicit Flows. These occur when tainted data influences control flow decisions:

```
// Tainted condition affects program state
if (tainted_variable) {
    clean_variable = secret_value; // Implicit leak
}
```

In this case, the value of `clean_variable` depends on the tainted condition, creating an implicit information flow even though no direct assignment from tainted data occurs.

Researchers have widely studied implicit flows, particularly pointer tainting. Slowinska and Bos [31] indicated basic limitations in tracking all implicit flows accurately. They found that attempts to handle them fully often lead to major false positives or "taint explosion" where the entire system becomes tainted.

Their analysis revealed that different approaches to pointer tainting face distinct tradeoffs. Limited Pointer Tainting generates many false positives for normal operations. Full Pointer Tainting results in uncontrolled taint spread throughout system data structures.

Similarly, Cavallaro et al. [8] analyzed the basic tradeoffs between analysis accuracy and performance overhead in dynamic taint analysis. They indicated that complete implicit flow tracking introduces major computing costs that limit practical use.

These findings indicate that practical taint analysis systems must make careful choices about which types of implicit flows to track. We must balance precision against usability and performance factors.

3 DESIGN AND IMPLEMENTATION

This chapter details the architecture and implementation of the dynamic taint analysis engine. The engine uses the dynamic instrumentation capabilities of the Frida framework to perform instruction level data flow tracking on the ARM64 architecture for both iOS and Android platforms.

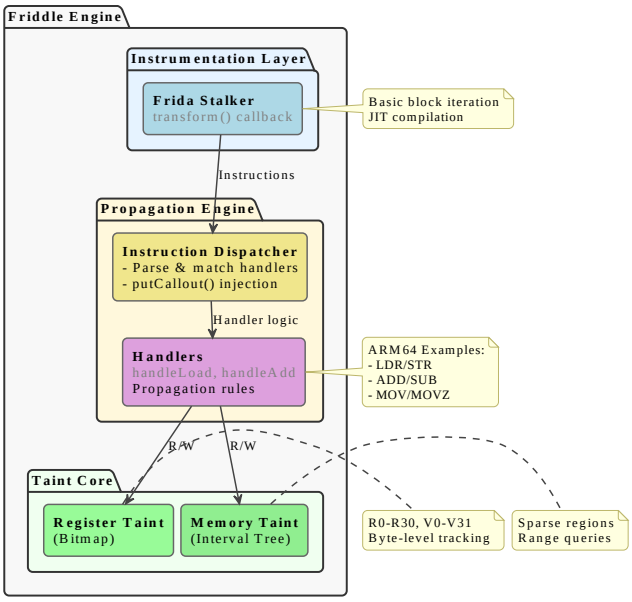


Figure 1: The layered, modular architecture of the Friddle taint analysis engine.

The system is designed with a layered, modular architecture that logically separates the analysis task into three cooperating components. This design improves system maintainability and testability. The chapter first describes the overall system architecture and then details the taint representation models and propagation logic.

3.1 System Architecture

The core architecture of the Friddle engine is based on the dynamic recompilation capability of Frida's Stalker. Its mechanism involves intercepting and rewriting the native instruction stream of a target process at runtime through JIT compilation, enabling fine grained monitoring of data flows. Friddle's core design was inspired by the experimental tool from Fioraldi for the x86/64 architecture, particularly the concepts of using bitmaps to track register taint and interval trees for memory taint [12]. Our main contribution is transforming this proof of concept into a functional and working framework on the mainstream mobile ARM64 architecture.

Our original work covers several areas. We designed and implemented parsing logic for ARM64 instructions and registers. We also developed precise propagation rules for a large number of common instructions. We covered architecture specific features like register aliasing, SIMD instructions, and the crypto extension. We refactored and optimized the core taint operation APIs and fixed an important bug in the interval tree implementation. This ensures that its merge and split operations work correctly when tainting or untainting memory regions. We also defined the project's modular architecture and developed debugging tools to aid analysis.

So, apart from the conceptual inspiration, the framework is our original work, from the low level instruction handling and core data structure fixes to the high level project architecture and the final cross platform engine implementation.

As illustrated in Figure 1, Friddle's system is designed with a layered, modular architecture, comprising the following core collaborating components:

Taint Core This component serves as the data foundation, responsible for storing and maintaining all taint marks at runtime. To balance efficiency and precision, it employs two specific data structures:
- **Register Taint:** A compact bitmap tracks which bytes within each general purpose and SIMD register are tainted.
- **Memory Taint:** An interval tree manages tainted regions sparsely distributed across the large memory space.

These two data structures, along with an API to operate on them, collectively form the Taint Core, providing the upper level engine with the capability to query and update the taint status.

Propagation Engine This engine is the core computational component responsible for dispatching instructions and applying taint propagation rules. It operates on the Taint Core's data structures to implement analysis logic. The engine parses instruction mnemonics and operands, then selects suitable handlers from a predefined set. For example, an `LDR` instruction matches with a `handleLoad` handler. Each handler contains specific taint propagation logic that updates the underlying Taint Core state at runtime.

Instrumentation Layer This layer directly uses Frida's Stalker to capture and rewrite native instructions at runtime. It coordinates with the Propagation Engine to inject analysis logic into the target program. Stalker processes new basic blocks through its `transform` callback, where Friddle iterates through each instruction and uses the `putCallout()` method to inject callout functions from the Propagation Engine. The instrumented basic block is then JIT compiled and cached by Frida, enabling instruction level analysis.

3.2 Taint Core

The Taint Core serves as the data foundation of the Friddle engine. It provides a unified interface for the upper level Propagation Engine to manipulate and query taint status. Due to the differences in physical characteristics and access patterns between registers and memory, we adapted distinct taint representation models for each. While the core concept of using bitmaps for registers and interval trees for memory was inspired by Fioraldi's x86 work, our ARM64 implementation needed complete redesign and development. This core consists of two specialized data structures: a Bitmap for registers and an Interval Tree for memory, along with an API to operate on them.

3.2.1 *Register Taint Representation.*

Granularity and Data Structure. We designed a byte level tracking granularity for registers. To implement this, we built a bitmap model for each physical register. In this model, an 8-bit bitmap describes the taint status of a 64-bit register, where each bit maps to one byte. This design allows us to accurately represent partial taint states, such as when an `ldrb w0, [src]` instruction taints only the lowest byte of the `w0` register.

Register Aliasing. Handling register aliasing in the ARM64 architecture is an important consideration in the design. For example, `x0` (64-bit) and `w0` (32-bit) share the same physical storage unit; similarly, SIMD registers `q0` (128-bit), `d0` (64-bit), and `s0` (32-bit) also share storage.

To address this, we employ a shared bitmap mechanism based on physical offset. During engine initialization, the `Registers` class iterates through predefined register metadata and maps all registers sharing the same physical storage (e.g., `x0` and `w0`) to the same Bitmap instance. When the engine operates on an aliased register (e.g., `w0`), its API, guided by the size defined in the metadata (32 bits, or 4 bytes), reads or writes only the corresponding first 4 bits of the shared Bitmap. This method ensures that modifications to the taint status of any aliased register are correctly reflected across all others, thus maintaining state consistency.

3.2.2 *Memory Taint Representation.*

Challenges and Data Structure. Given the scale of a program's memory address space, maintaining a separate taint flag for each byte of memory involves major space and time overhead. Existing approaches have observed that tainted data in memory often exists as contiguous regions, which provides a basis for optimization.

Following this approach, we use an Interval Tree to manage memory taint, similar to previous work. The Interval Tree stores address intervals `[start, end)` that represent contiguous tainted bytes rather than individual byte addresses. When a new taint interval is added, the tree automatically finds adjacent or overlapping existing intervals and merges them into larger intervals.

However, we found and fixed a critical bug in the original interval tree implementation where merge and split operations did not work correctly when tainting or untainting memory regions. We also developed additional API methods to better integrate the interval tree with our taint propagation engine.

Advantages. This data structure design offers two key advantages:

- **Space Efficiency:** Just one node represents a fully tainted 1MB buffer in the interval tree. Consequently, the storage overhead is proportional to the degree of fragmentation of tainted regions, not their absolute size.
- **Query Efficiency:** The properties of the interval tree data structure make its interval query performance more efficient than a linear scan, enabling rapid responses to `isTainted` or `isFullyTainted` checks.

3.2.3 *Core API and Propagation Patterns.*

The Taint Core provides a unified interface for the upper level Propagation Engine to manipulate taint status through a set of atomic APIs. These APIs contain the complex operations on the underlying Bitmap and Interval Tree structures. Key APIs include `spread(dest, src)` for direct register to register state copying, `union(bitmap1, bitmap2)` for merging taint from two sources, and pairs like `toBitMap`/`fromBitMap` and `toRanges`/`fromRanges` to convert between the register bitmap and memory interval representations. The Propagation Engine builds all complex instruction propagation rules by combining calls to these basic APIs.

3.3 Propagation Engine

The Propagation Engine is the core computational component of Friddle. We designed it to translate the static semantics of instructions into dynamic and concrete taint state transitions. The engine's architecture is determined by Frida Stalker's two phase execution model: a compile time instrumentation phase and a runtime execution phase. Friddle's instruction handling scheme is built entirely on this model of separating decision making (compile time) from execution (runtime), consisting mainly of a set of instruction handlers and a series of runtime callout functions.

3.3.1 *The Stalker Based Handler Callout Model.*

Compile time Decision by Instruction Handlers. Each handler we define, such as `handleLoadStoreSingleReg`, is a function that runs within the compile time `transform` callback. Its responsibility is to make decisions: the function inspects the current instruction's mnemonic and operands, and if they match its handling scope, it returns a reference to a specific callout function. This process does not perform any taint analysis itself but instead sets the logical path for the runtime analysis.

Runtime Execution by Callout Functions. A callout function is triggered at runtime. When the CPU executes a new basic block generated by the instrumentation and compilation layer, the embedded callout is activated. Its responsibility is to perform the analysis: it receives the runtime context (`ctx`) containing the current CPU state, calls operand parsing functions, applies preset propagation rules, and finally invokes the Taint Core's API to complete the taint state update.

In this way, the high overhead instruction matching and decision logic are executed only once when a basic block is first compiled. At runtime, the CPU directly jumps to the callout function containing the core analysis logic. This separation reduces redundant instruction matching overhead and contributes to overall analysis efficiency.

3.3.2 *Operand Parsing and Evaluation.*

To enable callout functions to efficiently access runtime data and parse complex Frida context structures, we implemented a set of operand parsing utilities. These functions, such as `parseRegOperand` and `parseMemOperand`, are responsible for extracting the concrete values of operands from the `ctx` context at runtime.

Our parsers fetch register values and calculate effective memory addresses. They also handle complex operand formats common in ARM64 instructions, such as register operands with extensions (`ext`) or shifts (`shift`). For instance, with the instruction `add x0, x1, x2, lsl #2`, when processing `x2`, `parseRegOperand` first reads its original value. Then it applies the `lsl #2` operation, and finally returns the computed result to the callout for taint propagation decisions. Listing 1 shows the core logic of these parsing utilities. The `parseMemOperand` function computes the final memory address. It also identifies indirect taint propagation, known as an implicit flow, by checking if any register in the address calculation is tainted.

```
// in taint_engine.js
function parseMemOperand(ctx, memOperand) {
    let addr = ptr(0);
    let isIndirectTainted = false;
```

```
    // Handle base register
    if (opVal.base) {
        let baseVal = readRegVal(ctx, opVal.base);
        addr = addr.add(baseVal);
        if (regs.isTainted(opVal.base)) {
            isIndirectTainted = true;
        }
    }
    // Handle index register (and shift/extension)
    if (opVal.index) {
        let indexVal = readRegVal(ctx, opVal.index);
        // ... apply shift or extension here ...
        addr = addr.add(indexVal);
        if (regs.isTainted(opVal.index)) {
            isIndirectTainted = true;
        }
    }
    // ...
    return { memAddr: addr, isIndirectTainted:
         isIndirectTainted, ... };
}

function parseRegOperand(ctx, operand) {
    let regName = operand.value;
    let computedVal = readRegVal(ctx, regName);
    if (operand.shift) {
        // ... shift computedVal based on operand.shift.type
             and .value ...
    }
    // ...
    return { name: regName, regVal: computedVal, ... };
}
```

Listing 1: Core Logic of Operand Parsing Functions

3.3.3 Typical Taint Propagation Rules. We define corresponding taint propagation rules based on the semantics of ARM64 instructions. Here are some examples for typical instructions:

- **Data Load Instruction (`ldr rd, [rn]`):** This represents data flow from memory to a register. The propagation rule is: if the taint state of the source memory address `[rn]` is not empty, this state is propagated to the destination register `rd`. We use `mem.toBitMap()` to convert the taint of a memory region into a bitmap, which is then applied to the destination register using `regs.fromBitMap()`.
- **Data Store Instruction (`str rn, [rd]`):** This represents data flow from a register to memory. The propagation rule is: if the taint state of the source register `rn` is not empty, this state is propagated to the destination memory address `[rd]`. We use `regs.toRanges()` to convert the register's taint bitmap into memory ranges, which are then updated into the memory's taint tree using `mem.fromRanges()`.
- **Arithmetic Instruction (`add rd, rn, rm`):** This is a data processing operation. Its propagation rule is defined as: the taint state of the destination register `rd` is the union of the taint states of the two source registers `rn` and `rm`. That is, T(rd) = T(rn) ∪ T(rm).
- **Taint Clearing Instruction (`mov rd, #imm`):** When a destination register is assigned an immediate value (a constant), it no longer carries any data from previous computations. Therefore, its propagation rule is to clear (`untaint`) all taint bits of the destination register `rd`.
- **Self-clearing Operations (`xor rd, rd, rd`):** Instructions like `xor` with the same register as all operands or `sub` with the same register always produce zero regardless of the register's value. Our engine detects these patterns and clears the destination register's taint state, preventing false propagation. When encountering `xor x0, x0, x0`, the handler recognizes that all operands refer to the same register and sets the taint bitmap of `x0` to zero.

3.3.4 Handling of Complex Instruction Semantics. A complete taint analysis engine must handle the complex instructions found in modern instruction sets. We designed and built specialized propagation rules for implicit data flows, Advanced SIMD, and cryptographic extension instructions in the ARM64 architecture.

Implicit Taint Propagation. Taint information can propagate not only through explicit data movement (like `MOV`) but also through implicit means, such as by influencing memory access addresses. We focus on handling the implicit propagation problem during address calculation, which is particularly important when dealing with table lookups common in real world algorithms.

- **Case Study: Base64 Encoding and AES S Box Lookups:** During Base64 encoding, input data is used as an index to look up the corresponding encoded character in a fixed character map. Similarly, in the SubBytes step of AES encryption, each byte in the state matrix is used as an index for a lookup in a permutation table called the S-Box. In both scenarios, if the data used as an index is tainted, the result "selected" from the table via this index should also be considered tainted.
- **Implementation:** Our engine captures this implicit flow by checking all registers involved in address calculation during memory operand parsing. In the `parseMemOperand` function, if the base or index register used to compute a memory address is itself tainted, the function's returned `isIndirectTainted` flag will be `true`. The callout function then uses this flag to execute propagation. For a table lookup operation like `ldrb w0, [x1, x2]`, the `isIndirectTainted` flag will be true as long as the index `x2` is tainted. This happens even if the data loaded from address `[x1 + x2]` is clean. The `ldrbRegAddrCallout` will then mark the destination register `w0` as tainted.

```
// in taint_engine.js
function ldrbRegAddrCallout(ctx) {
    // ...
    let dest = parseRegOperand(ctx, operands[0]).name;
    let { memAddr, isIndirectTainted } = parseMemOperand(ctx,
          operands[1]);

    if (isIndirectTainted) {
        // The address calculation itself is tainted;
             directly taint the destination.
        regs.taintWithOffsetAndSize(dest, 0, 1);
    } else {
        // Otherwise, propagate based on the memory's taint
             state.
        regs.fromBitMap(dest, mem.toBitMap(memAddr, 1));
```

```
    }
}
```

Listing 2: Handling of Implicit Taint in 'ldrb' Callout

Listing 2 shows the implementation of this implicit flow logic. The `ldrbRegAddrCallout` function first checks the `isIndirectTainted` flag. If true, the destination register `dest` is tainted directly because the address calculation was tainted. Otherwise, the function performs a standard propagation based on the memory's taint state.

Configurable Implicit Flow Tracking. Recognizing that many use cases do not require pointer tainting, Friddle provides configurable implicit flow tracking through the `globalState.enableImplicitTaint` flag. When disabled, the engine skips implicit flow detection and performs only explicit data propagation. This configuration option allows users to balance analysis precision against the risk of over tainting, providing flexibility for different analysis cases.

Advanced SIMD Instruction Set. The challenge with handling SIMD instructions is that their operations are executed in parallel on multiple "lanes" within a register. Therefore, taint tracking must be lane-aware. To do this, we built the specific propagation logic for these complex instructions. We referenced the official ARM Architecture Reference Manual, which details the precise behavior of each instruction, particularly for SIMD operations [4]. We use a `parseVector` utility function to parse Vector Arrangement Specifiers. We have implemented precise, lane-level propagation rules for various SIMD instructions.

- **Vector Table Lookup (`tbl Vd.8b, {Vn.16b}, Vm.8b`):** Whether the i-th lane of `Vd` is tainted depends on whether the index value in the i-th lane of `Vm` is tainted, or whether the data in `Vn` pointed to by that index is tainted. Our implementation follows the logic T(Vd[i]) = T(Vm[i]) ∪ T(Vn[Vm[i]]).
- **Vector Scalar Replicate (`dup Vd.4s, Wn`):** This instruction copies the value of a general purpose register `Wn` to all lanes of the destination vector `Vd`. Our implementation gets the 4-byte taint bitmap of the source register `Wn` and replicates it across the four single word lanes of `Vd`.
- **Vector Structured Load (`ld1 {v0.4s, v1.4s}, [x0]`):** This instruction loads data contiguously from memory address `[x0]` into multiple vector registers. Our code simulates this process, accurately reflecting the taint flow from a contiguous memory block to multiple vector registers.

Cryptographic Extension Instructions. Modern ARM processors include dedicated instructions (e.g., `aese`, `aesmc`) to accelerate symmetric encryption algorithms like AES. These instructions perform complex internal data permutations and mixing, making a bit-level simulation of their information flow extremely costly. We adopt a conservative and sound approximation. For an instruction like `aese Vd.16b, Vn.16b`, we simplify its data flow model to be the union of its inputs and outputs: if either the input `Vd` (previous state) or `Vn` (round key) is tainted, the output `Vd` register will be marked as fully tainted. This ensures that taint information is not lost during opaque cryptographic operations.

3.4 Instrumentation & Compilation Layer

The Instrumentation and Compilation Layer is the frontend interface and entry point for the Friddle engine, designed to serve as a bridge between the engine and the target program. Its core responsibility is to analyze and modify the original instruction stream during a compile time phase, injecting probes for subsequent runtime analysis. This works entirely based on the dynamic recompilation capabilities of Frida's Stalker.

3.4.1 Stalker's Dynamic Recompilation Workflow. Stalker's workflow begins by tracing a target thread. When the thread is about to execute a basic block (BB) that has not yet been instrumented, Stalker pauses the execution flow and first copies the instruction sequence of the original BB to a new memory region (a trampoline). All subsequent modifications and instrumentations are performed on this memory copy, ensuring the integrity of the original code.

After the copy is complete, Stalker invokes our `transform` callback, providing an `iterator` object that represents the instruction stream in the new memory region. We primarily use two methods from the `iterator` to construct the new basic block:

- `next()`: Allows us to access and inspect each original instruction in the BB one by one.
- `keep()`: Is responsible for writing the original instruction (after internal relocation by Stalker) into the final executable code, which is basic to preserving the program's original logic.

We perform instrumentation by using the core method `iterator.putCallout(callback)`. This method does not execute the `callback` function immediately. Instead, it inserts a call instruction to this JavaScript `callback` function into the new code stream being built. This `callout` function serves as the container for our runtime analysis logic. Finally, Stalker compiles this new, instrumented BB into executable code and replaces the entry point of the original BB with a jump instruction, redirecting the execution flow to our newly generated code. This process precisely implements the core mechanism of "compile time injection, runtime triggering."

3.4.2 Handler Dispatch Pattern. To avoid writing a single, monolithic logic within the `transform` callback to handle all instructions, we designed and implemented a structured Handler Dispatch Pattern. The classification of instructions into these handler categories was guided by the functional groupings presented in the ARMv8-A instruction set overview [26].

We maintain a handler list (`stalkerHandleList`) containing multiple handler functions. Within the `transform` callback, a dispatch loop iterates through this list, passing the current instruction to each handler function (e.g., `handleLoadStoreSingleReg`, `handleArithmeticImmediate`). Each handler is concerned with only a small category of instructions, which it identifies by checking the instruction's mnemonic and operands. If a match is found, it becomes responsible for calling `putCallout()` to inject the corresponding runtime analysis function.

This pattern decouples the processing logic for different instructions into their own independent modules, enhancing code modularity and extensibility. To add support for new instructions in the future, we only need to create a new handler function and register it to the list, without modifying existing logic.

3.4.3 Compile Time Visualization and Debugging Support. Debugging tools are needed when developing a complex taint analysis engine. To this end, we integrated a set of auxiliary features into the compile time phase to provide visualization and debugging support during development.

First, we implemented a mechanism to visualize the instruction handling status at compile time. After the `transform` callback dispatches an instruction, a function named `printInstrCompileDebugInfo` prints it to the console in different colors based on whether it was successfully matched by any handler. Unhandled instructions are displayed in white, while instrumented instructions are shown in yellow or other colors depending on the instrumentation type. This feature provides a clear view of our instruction coverage, allowing us to quickly identify unsupported instructions and guide future development.

Second, to trace the origin of a `callout` at the compile time stage, we implemented a custom wrapper function named `iteratorPutCalloutWrapper`. All calls to the native `iterator.putCallout()` are made through this wrapper. While injecting the `callout`, the wrapper uses JavaScript's error stack tracing capabilities to retrieve and print the name of the handler function that initiated the injection and its location in the source code. This is very convenient: when unexpected behavior occurs at runtime, we can use the compile time logs to directly locate which handler injected the analysis logic for the problematic instruction. This shortens the debugging cycle. Listing 3 shows an excerpt from this debugging mechanism in practice. The log clearly displays which handler function `handleArithmeticImmediate` processed each instruction e.g., `sub sp, sp, #0x60`. It also shows at which source file location `taint_engine.js:2605` the callout injection occurred, providing great convenience for developers.

Finally, we implemented a series of injectable debugging `callouts`. These functions (such as `printInstrDuringRuntime` and `showTaintStatusWhenUpdate`) are registered and injected through the handler-dispatch pattern, just like the core analysis `callouts`. They enable us to observe instruction execution, operand values, and taint status changes with fine granularity at runtime, providing important observability for verifying the correctness of propagation rules and troubleshooting complex scenarios.

```
0x7b5af3db74: sub sp, sp, #0x60 (3 reg reg imm)
     libandroidtest.so ...
  subRegRegImmCallout -> at handleArithmeticImmediate (
     taint_engine.js:2605)
0x7b5af3db78: stp x29, x30, [sp, #0x50] (3 reg reg mem)
     libandroidtest.so ...
  stpRegRegAddrCallout -> at handleLoadStorePair (
     taint_engine.js:2217)
0x7b5af3db7c: add x29, sp, #0x50 (3 reg reg imm)
     libandroidtest.so ...
  addRegRegImmCallout -> at handleArithmeticImmediate (
     taint_engine.js:2585)
[+] 0x7b5af3db80: mrs x8, tpidr_el0 (2 reg sys)
     libandroidtest.so ...
0x7b5af3db84: ldr x8, [x8, #0x28] (2 reg mem) libandroidtest
     .so ...
  ldrRegAddrCallout -> at handleLoadStoreSingleReg (
     taint_engine.js:1705)
0x7b5af3db88: stur x8, [x29, #-8] (2 reg mem) libandroidtest
     .so ...
  sturRegAddrCallout -> at
     handleLoadStoreSingleRegUnscaledOffset (taint_engine.js
     :2000)
...
[+] 0x7b5af3dba0: nop (0) libandroidtest.so ...
0x7b5af3dba4: adr x1, #0x7b5afedda0 (2 reg imm)
     libandroidtest.so ...
  adrRegImmCallout -> at handleAddressGeneration (
     taint_engine.js:2848)
0x7b5af3dba8: mov w0, #3 (2 reg imm) libandroidtest.so ...
  movRegImmCallout -> at handleMoveWideImmediate (
     taint_engine.js:2791)
```

Listing 3: A compile time log excerpt demonstrating source tracking for callout injections. It shows instructions being dispatched to handlers and reports the source location of each injection.

4 EVALUATION

In this chapter, we measure the effectiveness and performance of the Friddle engine. While previous chapters detailed Friddle's internal architecture and working principles, the goal of this chapter is to answer key questions about its real world performance through a series of experiments.

4.1 Evaluation Goals and Methodology

To ensure our evaluation has clear objectives and a structured approach, we build our experimental methodology around the three core research questions (RQs) introduced in §1. We systematically address each research question through targeted experiments:

To answer these questions, we use an experimental approach based on our custom benchmark suite, FriddleBench. The following subsections detail the experimental setup, the metrics used, and the final results and analysis.

4.2 Experimental Setup

This section details the hardware and software environment used for this evaluation and the evaluation process.

4.2.1 Hardware and Software Environment. To evaluate Friddle's cross platform capabilities, all our experiments were conducted on two mobile devices:

- **iOS Device (Jailbroken):** An iPhone 6s running iOS 15.8.3.
- **Android Device (Rooted):** A Google Pixel 6 running Android 12.

All dynamic instrumentation and analysis tasks were performed using Frida version 17.2.11.

4.2.2 FriddleBench: The Benchmark Suite. We designed and built a benchmark suite named FriddleBench. The core of this suite is a series of C functions that simulate different data leakage scenarios. These functions are built into standard Android and iOS applications, providing users with a unified interface and testing workflow. The suite includes the following core test logic:

- **B1: Plaintext Handling**

Description: A C function that copies its input and returns the data, simulating a basic data flow.

- **B2: Base64 Encoding**
  Description: A C function that performs Base64 encoding on the input data. This process involves many bitwise operations and table lookups, designed to test implicit flow tracking.
- **B3: Custom AES Encryption**
  Description: A C function that encrypts input data using our own implementation of the AES-128 algorithm, which does not rely on hardware acceleration. This tests the ability to track complex algorithmic logic.
- **B4: System Library Encryption**
  Description: This C function calls a third party crypto library to process data. On Android, it uses our self compiled BoringSSL library; on iOS, it calls the native CommonCrypto (`CCCrypt`) framework. This test checks Friddle's ability to track taint propagation through third party libraries that are heavily optimized with SIMD instructions.

False Positive Mode. To verify that Friddle does not produce false positives, we added a false positive mode for each benchmark test. In this mode, the program creates a clean buffer filled with character 'A' that has the same length as the expected output. The source function still processes the real user input through the complete transformation (copy, Base64 encode, or AES encrypt), so taint spread happens during processing. However, the sink function receives the clean buffer instead of the processed result. If Friddle works correctly, it should detect that the sink parameters are not tainted, even though tainted data was created during processing.

This simulates real world scenarios where applications may use user data for computation but do not actually leak it. This design tests whether our framework can avoid the "taint explosion" problem where all data would be wrongly marked as tainted.

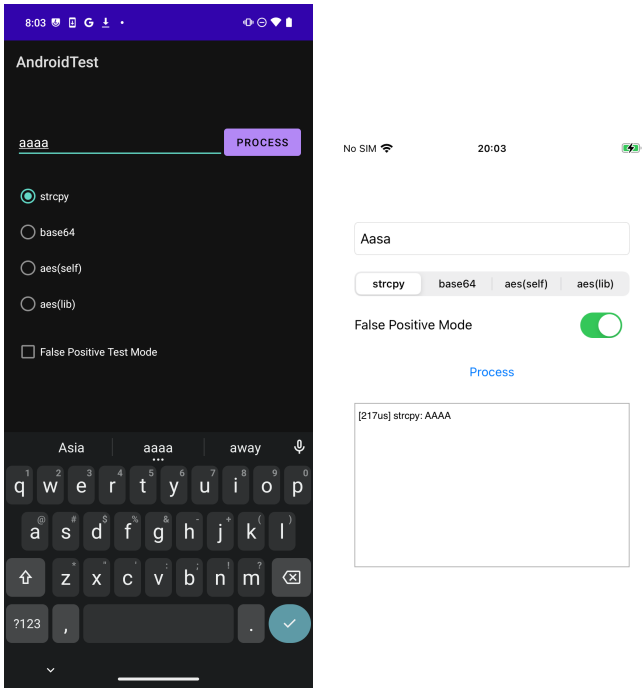
4.2.3 Evaluation Process. To provide a unified testing experience and ensure consistency, our Android and iOS benchmark applications share the same design, as shown in Figure 2. The app provides an input field, allowing a user to enter any string as sensitive data. The user can then select one of four test modes (corresponding to B1-B4) to start the test.

Regardless of the chosen mode, the underlying data processing follows the same call sequence:

(1) The program first calls a generic `source` function, passing the user provided data from the input field as an argument.
(2) Inside the `source` function, the corresponding benchmark function (one of B1-B4) is called to process the data based on the user's selection.
(3) After processing, the result is passed to a generic `sink` function.

Based on this workflow, we measure correctness and performance:

- **Correctness Evaluation (Answering RQ1):** We evaluate both true positive and false positive detection. For true positive tests, we start the taint engine when the `source` function is called and examine the `sink` function arguments to determine if they are correctly marked as tainted. For false positive tests, we use clean data at the sink while processing tainted data in between, and verify that the sink receives untainted parameters. If the taint status meets expectations in both cases, the detection is considered successful.
- **Performance Evaluation (Answering RQ2):** We measure the performance overhead by recording the total time from before the `source` function is called until after the `sink` function has completed. We record a start timestamp just before calling `source` and an end timestamp just after calling `sink`. The difference between these timestamps, measured in milliseconds, represents the execution time. We compare the time taken with Friddle enabled against the time for native execution (no instrumentation) to calculate the overhead.
- **Instruction Coverage Evaluation (Answering RQ3):** We analyze the distribution of ARM64 instructions in Android system libraries to check Friddle's support coverage. We collect instruction usage data and calculate the coverage rate that our framework supports.



(a) FriddleBench on Android. **(b) FriddleBench on iOS.**

Figure 2: The user interface of the FriddleBench benchmark applications on both platforms.

4.3 API Usage and Analysis Configuration

We use the JavaScript API provided by Friddle to configure and start taint analysis for all benchmarks. As shown in Listing 4, we use Frida's `Interceptor` to hook target functions. This listing presents the actual script used for analysis on the Android platform, and the `console.log` statements within it correspond to the output shown in our results in §4.4.

```
import { init_taint, start_taint, stop_taint } from "./
    taint_engine.js";

// 1. Define the target native library
const MODULE_NAME = "libandroidtest.so";
const targetModule = Process.findModuleByName(MODULE_NAME);

// 2. Find the entry points for analysis
const processData = targetModule.findExportByName(
   "Java_com_friddle_androidtest_MainActivity_nativeProcess
       "
);
const source = targetModule.findSymbolByName("
    _ZL11source_modePKcmPcPmi");
const sink = targetModule.findSymbolByName("
    _ZL9sink_modePcmi");

// 3. Hook the main function to initialize and start the
    engine
Interceptor.attach(processData, {
   onEnter: function (args) {
      console.log(`processData at ${targetModule.name} ${
          targetModule.base} ${this.context.pc}`);
      init_taint(source, sink);
      start_taint();
   },
});

// 4. Hook the sink function to stop analysis and report
    results
Interceptor.attach(sink, {
   onEnter: function (args) {
      console.log(`sink hit at ${targetModule.name} ${
          targetModule.base} ${this.context.pc}`);
      stop_taint();
   },
});
```

Listing 4: The actual script for configuring Friddle analysis on Android.

4.4 Results and Analysis

This section presents and analyzes the experimental data collected from the FriddleBench suite to answer our three core research questions (RQ1: Correctness, RQ2: Performance, and RQ3: Instruction Coverage).

4.4.1 Correctness Analysis (Answering RQ1). To answer RQ1, we check whether Friddle detects taint propagation from the `source` to the `sink` function in each benchmark. We define a successful detection as an instance where Friddle's report demonstrates that the sensitive data parameter received by the `sink` function is marked as tainted.

Table 1 summarizes the results of this verification.

The results indicate that Friddle detected taint propagation in all designed test scenarios. This provides a basis for evaluating the correctness of its core functionality. The result of each test corresponds to key technical points discussed in our Design and Implementation chapter:

Table 1: Correctness Verification of Friddle on FriddleBench.

ID	Benchmark	Platform	Detected?
B1	Plaintext	Android & iOS	Yes
B2	Base64	Android & iOS	Yes
B3	Custom AES	Android & iOS	Yes
B4	Library AES	Android & iOS	Yes

- The result of **B1 (Plaintext Handling)** confirms the engine's tracking capability in a basic data flow scenario.
- The result of **B2 (Base64 Encoding)** indicates that Friddle can handle implicit taint propagation introduced by table lookup operations. This aligns with the propagation rules we designed for `TBL` and other lookup instructions in §3.3.4.
- The result of **B3 (Custom AES Encryption)** indicates that our propagation model for standard arithmetic and logical instructions can cover complex algorithmic logic that does not rely on hardware acceleration.
- The result of **B4 (System Library Encryption)** shows that Friddle can track taint propagation through highly optimized library functions. This corresponds to the logic we implemented in §3.3.4 for managing taint state in SIMD instructions, which is designed to handle hardware cryptographic instructions such as `AESE` and `AESMC`.

To provide direct evidence, Listing 5 shows the raw log output from Friddle during the execution of the B2 (Base64 Encoding) test. The log shows that when program flow reaches the `sink` function, Friddle reports that the `output_buffer is fully tainted`, which aligns with our definition of a successful detection.

```
Taint started at ins *** 0x7378d02b74 : sub sp, sp, #0x60
    ***
source -> x0: 0xb4000074053ff790, x1: 0x3, ...
mem tainted:
0xb4000074053ff790,0xb4000074053ff793
...
Taint stopped at ins *** 0x7378d02d48 : ldr x16, ...
sink -> x0: 0x7378db25a0, x1: 0x4
output_buffer is fully tainted
mem tainted:
0x7378db25a0,0x7378db25a4,0x7fd5a0854c,[...]
regs tainted:
x28(1.1.1.1.1.0.0.0),w28(1.1.1.1)
...
sink hit at libandroidtest.so 0x7378c80000 0x7378d02d48
```

Listing 5: Raw log output from Friddle showing successful detection in the B2 benchmark.

False Positive Analysis. To verify that Friddle does not produce false positives, we ran false positive tests for each benchmark scenario using the design described above. Table 2 shows the results of this verification.

Table 2: False Positive Verification of Friddle on FriddleBench.

ID	Benchmark	Platform	Clean Data Detected?
B1-FP	Plaintext	Android & iOS	Yes
B2-FP	Base64	Android & iOS	Yes
B3-FP	Custom AES	Android & iOS	Yes
B4-FP	Library AES	Android & iOS	Yes

The results show that in all false positive test scenarios on both platforms, Friddle correctly determined that the sink function received clean (not tainted) data. This demonstrates that our framework can distinguish tainted and clean data, avoiding the "taint explosion" problem where everything would be wrongly marked as tainted. The success of these tests confirms that our taint propagation rules are correct and that Friddle does not create false alarms when no actual data leakage occurs.

To provide concrete evidence for false positive detection on iOS, Listing 6 shows the raw log output from Friddle during a false positive test execution. The key result shows `output_buffer is not tainted`, confirming that Friddle correctly identified that the sink received clean data despite processing tainted data during the intermediate steps.

```
Taint started at ins *** 0x104e50fa0 : stp x28, x27, [sp,
    #-0x20]! ***
source -> x0: 0x28286e7c8, x1: 0x3, x2: 0x104e616b0, x3: 0
    x16afb0dc8
mem tainted:
0x28286e7c8,0x28286e7cb
...
Taint stopped at ins *** 0x104e512c0 : ldr x16, #0x104e512c8
     ***
sink -> x0: 0x104e61eb0, x1: 0x3
output_buffer is not tainted
mem tainted:
0x104e616b0,0x104e616d0,0x11e0b4008,0x11e0b400b,[...]
regs tainted:
x20(0.0.0.0.0.0.0.1),x22(0.0.0.0.0.0.0.1),x25
    (1.1.1.1.1.1.1.1),w25(1.1.1.1)
...
sink hit at iostest 0x104e4c000 0x104e512c0
```

Listing 6: Raw log output from Friddle showing successful false positive detection on iOS.

4.4.2 Performance Overhead Analysis (Answering RQ2). To answer RQ2, we measured the execution time of each benchmark under two conditions: 1) native execution (without Friddle enabled); and 2) with Friddle enabled for analysis. We use the Slowdown Factor to quantify the performance overhead, calculated as follows:

$$\text{Slowdown Factor} = \frac{\text{Execution Time with Friddle}}{\text{Native Execution Time}}$$

We ran each benchmark five times on both Android and iOS and used the average execution time for our calculations. Tables 3 and 4 present the performance data from the two platforms.

Table 3: Performance Overhead of Friddle on iOS.

ID	Benchmark	Native (ms)	Friddle (ms)	Slowdown
B1	Plaintext	0.010	377.57	≈37,016x
B2	Base64	0.014	455.65	≈32,546x
B3	Custom AES	0.047	3,093.74	≈65,824x
B4	Library AES	0.040	1,440.42	≈36,010x

Table 4: Performance Overhead of Friddle on Android.

ID	Benchmark	Native (ms)	Friddle (ms)	Slowdown
B1	Plaintext	0.073	711.96	≈9,726x
B2	Base64	0.067	741.15	≈11,095x
B3	Custom AES	0.093	2,066.25	≈22,313x
B4	Library AES	0.077	1,072.12	≈13,887x

The experimental data shows that enabling Friddle introduces major performance overhead, with slowdown factors in the range of 9,700x to 65,800x. This overhead is directly related to the DBI technique and the instruction level granularity of our analysis. The causes can be attributed to the following points:

- **Short Native Execution Time:** The benchmark functions have native execution times on the order of microseconds (µs). For such short running targets, the one time setup and JIT compilation costs from Frida/Stalker dominate the total execution time, leading to a high relative slowdown factor.
- **JIT Compilation and Callout Overhead:** This is a primary component of the performance overhead. Frida's Stalker operates on a per basic block (BB) basis. The first time a BB is encountered, Stalker performs JIT compilation and injects our JavaScript analysis logic (the callout). While the compiled BB can be reused, subsequent executions of instrumented instructions still require a context switch from the native code (trampoline) to the JavaScript engine and back. This switching process, along with the execution of the JS function itself, has a much higher cost than executing native instructions directly.
- **JavaScript Engine Overhead:** All taint propagation rules and state updates are performed in JavaScript. As a dynamic language, its execution is relatively less efficient compared to compiled native code.

From a cross benchmark comparison, we can also note two points:

- **B3 (Custom AES)** exhibits the highest performance overhead on both platforms. This is because our custom implementation contains a high density of standard arithmetic and logical instructions, leading to the largest number of instrumented instructions and context switches.
- The overhead of **B4 (System Library Encryption)** is lower than B3. This is because optimized libraries like BoringSSL and CommonCrypto use SIMD instructions. A single SIMD instruction (e.g., `AESE`) can perform the work of multiple

standard instructions. This reduces the total number of instructions that need to be instrumented and thus the total number of context switches.

In summary, the observed performance overhead is a direct tradeoff for the instruction level granularity and cross platform flexibility that Friddle provides. This makes the tool suitable for its intended use case of in depth analysis of specific code segments during security auditing, rather than for always on production monitoring where performance is a primary concern.

4.4.3 Instruction Coverage Analysis (Answering RQ3). To check Friddle's support for the ARM64 instruction set, we did a full analysis of instruction distribution in Android system libraries. This analysis helps us understand what our framework can cover and what limits it has.

Analysis Method. We analyzed instruction data taken from 635 ARM64 Android system shared libraries (.so files) extracted from the system/lib64 directory. The raw dataset had 25,053,202 instruction occurrences with 510 different instruction patterns. An instruction pattern represents a unique combination of instruction mnemonic, operand count, and operand types. For example, `ldr reg mem` (load register from memory) and `ldr reg mem imm` (load register from memory with immediate offset) are different patterns even though both use the `ldr` instruction. To focus on data flow analysis, we removed control flow and comparison instructions, as well as undefined instructions (UDF), which do not affect taint spread but make up a large part of the instruction stream. After removing 7,233,299 instructions (28.9%) with 33 patterns (6.5%), we analyzed 17,819,903 data flow related instructions covering 477 instruction patterns.

Coverage Results. Table 5 shows our coverage analysis results with pattern examples. Our analysis shows that we support 134 out of 477 instruction patterns (28.1% pattern coverage), but these supported patterns account for 98.5% of actual instruction execution frequency (17,549,908 out of 17,819,903 instructions).

As shown in the table, our framework supports key data flow patterns across all major handler categories while not supporting non data flow patterns that do not affect taint analysis. The supported patterns cover the basic operations needed for taint analysis in real world applications, from basic memory operations to crypto extensions. The unsupported patterns mainly consist of system control, debug, security, and comparison operations that do not affect data flow.

Analysis and Implications. The results indicate that while we support only about one third of instruction pattern types, we cover 98.5% of actual instruction runs. This means that our framework focuses on the high frequency, core data flow instruction patterns that are most important for taint analysis.

As shown in Table 5, we support key patterns across all major handler categories. Memory Access patterns like `ldr reg mem` and Arithmetic Logical patterns like `add reg reg imm` are the most frequent and form the base of our coverage. We also support advanced patterns including Vector Operations (`dup reg_v reg`) and Extensions (`aese reg_v reg_v`), which are important for analyzing modern applications that use optimized libraries.

The remaining 1.5% of unsupported instructions mainly consist of control flow, system, and debug patterns that do not take part in data flow and have no impact on taint tracking.

This coverage pattern confirms our design choice to focus on data flow related instruction patterns organized by our handler structure rather than trying to cover all patterns. The 98.5% execution frequency coverage indicates that the most frequent data flow patterns (`ldr`, `str`, `mov`, `add`) are all supported. The unsupported patterns are distributed across many low-frequency, specialized instructions rather than concentrated in common application hot paths.

5 DISCUSSION AND FUTURE WORK

In this chapter, we analyze the evaluation results in greater depth and discuss the architectural tradeoffs behind Friddle's performance. We systematically outline the limitations of our current implementation and propose clear directions for future work.

5.1 Architectural Tradeoffs and Design Philosophy

The evaluation results show that Friddle introduces major performance overhead during runtime. This outcome is not a design flaw but a direct reflection of our architectural choices and design philosophy. In this research phase, our primary goal was to prove the feasibility and correctness of instruction level, cross platform taint analysis; performance optimization was not a priority.

We chose to build on Frida and its JavaScript API to maximize development speed, simplify debugging, and ensure cross platform compatibility. This decision is fundamentally a tradeoff: we sacrificed runtime performance for higher development speed and broader platform support. As analyzed in the evaluation chapter, the sources of this overhead are consistent: the JIT compilation cost of the Frida/Stalker engine, the frequent context switching between native code and our JavaScript analysis logic, and the execution overhead of the JavaScript engine itself.

Therefore, Friddle's current performance profile defines its core use case: deep and precise data flow analysis of small, high risk code paths during **development and security auditing**. In such scenarios, analytical accuracy, flexibility, and ease of deployment are far more critical than raw performance. This tradeoff is reasonable and appropriate for a tool intended for analysis and auditing.

5.2 Limitations and Future Work

Beyond performance, our current implementation of Friddle has other limitations. These limitations point to clear directions for future research and improvement.

- **Single threaded Tracing:**
  Limitation: Our current implementation of Friddle only supports tracing a single thread. If a target application processes or passes tainted data across multiple threads, data flows occurring in threads other than the one being traced will be missed, leading to false negatives.

Table 5: Instruction Pattern Coverage Analysis with Examples.

Category	Pattern Example	Description	Usage Example	Supported	Freq %
Supported Data Flow Patterns (134/477 patterns, 98.5% execution frequency)					
Memory Access	`ldr reg mem`	Load register from memory	Basic data loading	Yes	16.68
Data Processing	`mov reg reg`	Move register to register	Data transfer	Yes	15.41
Arithmetic Logical	`add reg reg imm`	Add register and immediate	Address calculation	Yes	12.08
Memory Access	`str reg mem`	Store register to memory	Basic data storing	Yes	7.84
Memory Access	`ldp reg reg mem`	Load pair of registers	Structure loading	Yes	4.50
Arithmetic Logical	`and reg reg imm`	Bitwise AND with immediate	Bit manipulation	Yes	0.83
Arithmetic Logical	`mul reg reg reg`	Multiply two registers	Mathematical operations	Yes	0.28
Floating Point SIMD	`fmul reg reg reg`	Floating point multiply	FP operations	Yes	0.19
Data Processing	`bfi reg reg imm imm`	Bit field insert	Bit field operations	Yes	0.11
Floating Point SIMD	`fadd reg_v reg_v reg_v`	Vector floating point add	Vector arithmetic	Yes	0.06
Vector Operations	`dup reg_v reg`	Duplicate scalar to vector	SIMD data setup	Yes	0.03
Extensions	`aese reg_v reg_v`	AES encrypt single round	Crypto operations	Yes	<0.01
Extensions	`aesmc reg_v reg_v`	AES mix columns	Crypto transformations	Yes	<0.01
Vector Operations	`tbl reg_v reg_v reg_v`	Table lookup	SIMD table operations	Yes	<0.01
Unsupported Non Data Flow Patterns (343/477 patterns, 1.5% execution frequency)					
System Control	`mrs reg other`	Move system register	System state access	No	0.56
No Operation	`nop`	No operation	Padding/alignment	No	0.08
FP Comparison	`fcmp reg reg`	Floating point compare	Conditional logic	No	0.07
Debug/Halt	`hlt imm`	Halt with immediate	Debug/breakpoint	No	0.04
Authentication	`autiasp`	Authenticate return address	Security/pointer auth	No	0.01
Authentication	`paciasp`	Pointer auth code	Security/pointer auth	No	0.01
Branch Hint	`bti`	Branch target identification	Control flow integrity	No	0.01
System Control	`dmb other`	Data memory barrier	Memory synchronization	No	0.01

Future Work: Future work will extend Friddle to support multithreaded analysis. This will require designing a mechanism to monitor multiple threads simultaneously and correctly handle taint propagation during inter thread data exchange.

- **Instruction Coverage and Instrumentation Stability:**
  Limitation: The ARM64 instruction set is extensive and complex. While we have covered many common instructions, achieving complete coverage is a major challenge. For unhandled instructions, our engine defaults to not propagating taint, which can lead to false negatives. Furthermore, during testing, we observed that instrumenting certain atomic instructions (e.g., `stlxrh`, `stxr`, `stlxr`) can cause the target application to enter an infinite loop. We observed similar instability on iOS when tracing deep into the core system library `libdispatch.dylib`. This may be due to a bug in Frida's instrumentation of these specific instructions.
  Future Work: Future work will focus on improving the engine's robustness. This includes developing more precise handlers for known problematic instructions. We also plan to explore using an Intermediate Representation (IR) to speed up the implementation of propagation rules and systematically improve instruction coverage. By converting native instructions to an IR, we can implement taint logic on a smaller set of abstract operations, rather than for each specific instruction.
- **Performance Optimization:**
  Limitation: The current implementation, based entirely on JavaScript, can be greatly improved for performance.
  Future Work: Future optimizations can be approached in two parts. First, we can migrate performance critical taint propagation logic from JavaScript callbacks to C modules, which Frida also supports. This would greatly reduce context switching overhead. Second, we can implement function level summaries for common library functions (e.g., `memcpy`) to bypass expensive instruction level tracking.
- **Expanding Analysis Capabilities:**
  Future Work: Two important extensions are planned. The first is to support a **multilabel taint system**, allowing us to assign different labels to different taint sources (e.g., user input, GPS location) for more fine grained analysis. The second is to implement **taint path reconstruction**, which would log and show the full propagation path from source to sink, providing useful information for security audits.

6 RELATED WORK

This chapter reviews information flow tracking and taint analysis techniques on mobile platforms. By analyzing existing research, this chapter aims to show how Friddle relates to existing work.

6.1 Taint Analysis on Android

As the mobile operating system with the highest market share, Android's security analysis techniques, particularly taint analysis, have been widely studied.

6.1.1 Static Taint Analysis. Static taint analysis examines program code before execution to identify potential data leaks. `FlowDroid` is a representative static analysis tool that provides high precision taint analysis for Android applications [5]. `FlowDroid` accurately models the lifecycle and callback functions of Android applications. Compared to earlier tools, it is more precise in handling aliasing and program context, which reduces incorrect detections [17, 38].

Although static analysis can cover all code paths, it faces challenges when analyzing native code. For instance, studies like `JN-SAF` and `JuCify` attempt to analyze the interaction between Java and native code but encounter issues with path explosion and the complexity of constraint solving [29, 35]. One study indicates that the practicality of existing static analysis tools is limited when dealing with large, complex applications [25]. These limitations show the need for dynamic analysis methods as a complement.

6.1.2 Dynamic Taint Analysis. Dynamic taint analysis monitors data flows during actual program execution to achieve more precise results.

Dynamic Analysis at the Java Layer. The pioneering work in this area is `TaintDroid`, which provides real time taint tracking by modifying Android's Dalvik virtual machine [11]. As Android transitioned to the ART environment, subsequent tools evolved accordingly. For example, `TaintART` modified the ART compiler to support taint analysis [33], while `TaintMan` enabled analysis on non rooted devices by repackaging applications [39]. Other research, such as `ViaLin`, uses static analysis to guide dynamic analysis to reduce performance overhead [2]. A common characteristic of these tools is that their analysis scope is primarily limited to the Java code layer, making them unable to effectively track data flows that enter native code.

Dynamic Analysis for Native Code. Native code is widely used in Android applications. Studies show that many benign and malicious applications contain native code. Malicious software often uses native code to hide malicious behavior [1, 35–37]. Therefore, the analysis of native code is important.

Some studies have attempted to address this issue. `NDroid` aims to track data flows across both Java and native layers [36]. `JNFuzz-Droid` combines fuzzing with dynamic taint analysis to explore more execution paths in native code [7]. While these tools have made progress in analyzing Android native code, they do not support the iOS platform.

6.1.3 Hybrid Analysis Methods. Hybrid analysis methods combine the advantages of static and dynamic analysis. These approaches typically use static analysis to identify suspicious code paths, and then use dynamic execution to verify whether data leaks actually exist [27]. This helps improve analysis efficiency and accuracy. However, most hybrid methods still focus primarily on Java code and have not fully addressed the problem of analyzing native code.

6.2 Taint Analysis on iOS

Compared to Android, there is less public research on taint analysis for the iOS platform. This is mainly because iOS is a closed system. Existing analysis work for iOS is mostly conducted at the function level, for example, by hooking Objective-C methods to monitor high level data flows. These methods cannot look inside functions to track data operations at the instruction level. Therefore, performing instruction level information flow analysis for native code on the iOS platform remains an area that requires further research.

6.3 Frameworks on Other Platforms

Several established taint analysis frameworks exist for other platforms, but they cannot be directly adapted for mobile native code analysis due to basic architectural and platform differences.

Minemu. Minemu [6] provides high performance taint tracking for x86/x64 architectures. However, it cannot be adapted for mobile platforms because: (1) it is specifically designed for x86 instruction semantics and memory models; (2) it requires extensive kernel modifications that are incompatible with mobile OS security models; and (3) it lacks support for ARM64 specific features like Advanced SIMD and cryptographic extensions.

libdft. libdft [22, 23] offers dynamic data flow tracking through Intel Pin DBI. Its limitations for mobile use include: (1) dependence on Intel Pin, which does not support ARM64 mobile processors; (2) x86 specific instruction handling that would require complete redesign for ARM64; and (3) lack of support for mobile platform specific challenges like cross language (Java-Native) data flows.

Architecture and Platform Challenges. The basic challenge is that existing frameworks target server/desktop x86 environments with different: (1) instruction set architectures; (2) calling conventions and register usage; (3) memory management models; and (4) system call interfaces. Mobile platforms additionally introduce unique challenges like app sandboxing, code signing, and cross language runtime environments that existing tools do not address.

Existing research has made major progress in taint analysis on the Android platform, covering static, dynamic, Java, and native code layers. However, most tools do not support cross platform analysis, and there is a gap in instruction level dynamic analysis for the iOS platform. The design goal of `Friddle` is to address these issues by providing a unified, cross platform dynamic taint analysis framework for native code.

7 CONCLUSION

This study aimed to solve the problem of information flow tracking in the native code of mobile applications. By designing and implementing the Friddle framework, we demonstrated that dynamic taint analysis at the instruction level is an effective cross platform tracking method that does not require modifying the application or the operating system. The successful operation of Friddle on both Android and iOS devices validates the viability of this technical approach.

In our research, we identified a core tradeoff: the balance between analysis precision, cross platform capability, and performance overhead. The evaluation results show that while Friddle achieves accurate tracking, it introduces major performance overhead. This is a direct consequence of its fine grained instruction monitoring. This characteristic makes the framework suitable for in-depth data flow analysis of specific code modules during development and security auditing, where analysis accuracy is the primary consideration.

We acknowledge the limitations of the current work, which also provide directions for future research. First, the framework currently supports only single threaded analysis and cannot handle data exchange in multithreaded applications, which can lead to incomplete results. Future work needs to develop a mechanism to support multithreaded analysis. Second, our coverage of the ARM64 instruction set is incomplete, and unhandled instructions can break the taint propagation. Systematically improving instruction coverage is a key task.

There is also room for improvement in performance and functionality. To reduce performance overhead, the core analysis logic could be migrated from JavaScript to a C module, or function summaries could be implemented for common library functions. Functionally, a multilabel taint system could be added to distinguish different data sources, or taint path logging could be implemented to provide more detailed analysis reports.

In summary, through the Friddle framework, we have presented a practical method for analyzing data flows in native code on mobile devices and have clarified its technical characteristics and limitations. We believe this work provides a reference for future related research and an extensible base platform that can contribute to the development of more advanced analysis techniques.

REFERENCES

[1] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo De Geus, Christopher Kruegel, Giovanni Vigna, et al. 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium 2016*. 1–15.

[2] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. 2023. ViaLin: Path-aware dynamic taint analysis for Android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1598–1610.

[3] Arm Limited. 2024. A64 Instruction Set Architecture. https://developer.arm.com/Architectures/A64%20Instruction%20Set%20Architecture.

[4] Arm Limited. 2024. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0602/latest/.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[6] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The world's fastest taint tracker. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 1–20.

[7] Jianchao Cao, Fan Guo, and Yanwen Qu. 2024. Jnfuzz-droid: A lightweight fuzzing and taint analysis framework for android native code. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 255–266.

[8] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. 2008. On the limits of information flow techniques for malware analysis and containment. In *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 143–163.

[9] checkra1n team. 2021. checkra1n. https://checkra.in.

[10] DynamoRIO. 2024. DynamoRIO Dynamic Instrumentation Tool Platform. https://dynamorio.org.

[11] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[12] Andrea Fioraldi. 2018. A proof-of-concept dynamic taint analysis tool for x86 and x86-64. https://github.com/andreafioraldi/taint-with-frida.

[13] Frida. 2024. Frida Dynamic instrumentation toolkit. https://frida.re.

[14] Frida. 2024. frida-gum: Cross-platform instrumentation and introspection library written in C. https://github.com/frida/frida-gum.

[15] Frida. 2024. Stalker - Frida Documentation. https://frida.re/docs/stalker/.

[16] FTC. 2023. Ovulation Tracking App Premom Will be Barred from Sharing Health Data for Advertising Under Proposed FTC Order. https://www.ftc.gov/news-events/news/press-releases/2023/05/ovulation-tracking-app-premom-will-be-barred-sharing-health-data-advertising-under-proposed-ftc.

[17] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. Scandroid: Automated security certification of android. (2009).

[18] Google. 2024. Android NDK. https://developer.android.com/ndk.

[19] Google. 2024. Android Open Source Project. https://source.android.com.

[20] Google. 2024. Android Runtime (ART) and Dalvik. https://source.android.com/docs/core/runtime.

[21] Intel. 2024. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[22] Vasileios P Kemerlis. 2022. libdft: Dynamic data flow tracking for the masses. *Rn* 1 (2022), R2.

[23] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 121–132.

[24] palera1n team. 2024. palera1n. https://palera.in.

[25] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do android taint analysis tools keep their promises?. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 331–341.

[26] Princeton University. 2021. An Overview of the ARMv8-A Instruction Set. https://www.cs.princeton.edu/courses/archive/spring21/cos217/reading/ArmInstructionSetOverview.pdf.

[27] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in Android applications that feature anti-analysis techniques.. In *NDSS*.

[28] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, Serge Egelman, et al. 2018. "Won't somebody think of the children?" examining COPPA compliance at scale. In *The 18th Privacy Enhancing Technologies Symposium (PETS 2018)*.

[29] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*. 1232–1244.

[30] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[31] Asia Slowinska and Herbert Bos. 2009. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *EuroSys*. http://www.cs.vu.nl/~herbertb/papers/pointless_eurosys09.pdf

[32] Statcounter. 2025. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[33] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 331–342.

[34] The Apple Wiki. 2024. checkm8 Exploit. https://theapplewiki.com/wiki/Checkm8_Exploit.

[35] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1137–1150.

[36] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 2018. Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 814–828.

[37] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards {On-Device}{Non-Invasive} Mobile Malware Analysis for {ART}. In *26th USENIX Security Symposium (USENIX Security 17)*. 289–306.

[38] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on android with static taint analysis. In *2012 third world congress on software engineering*. IEEE, 101–104.

[39] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2017. Taintman: An art-compatible dynamic taint analysis framework on unmodified and

non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing* 17, 1 (2017), 209–222.

A ARTIFACTS

A.1 Abstract

The paper presents Friddle, a cross platform instruction level dynamic taint analysis framework for mobile native code. We indicate Friddle's capability to track information flow in native code on both Android and iOS platforms, including complex scenarios such as implicit flows and cryptographic operations. This artifact contains the complete Friddle framework source code, test applications, and scripts needed to reproduce the evaluation results presented in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns. Friddle requires privileged access to mobile devices for dynamic instrumentation. Android devices need root access to run frida-server with system privileges. iOS devices require jailbreaking to bypass Apple's code signing restrictions. Evaluators should use dedicated test devices and avoid using these devices for personal data or production environments.

A.2.2 How to access. The source code can be accessed by https://github.com/the7urn1ph34d/friddle.

A.2.3 Hardware dependencies.

- **Android:** ARM64 based Android device with unlockable bootloader
- **iOS:** iPhone or iPad with A10 or earlier processor (compatible with checkm8 jailbreak). iOS 15 recommended (tested on iOS 15.8.3)
- **Host Machine:** x86_64 or ARM64 Linux/macOS development machine with minimum 8GB RAM

A.2.4 Software dependencies.

- **Operating System:** Ubuntu 24.04 LTS, macOS 15 Sequoia, or Windows 11
- **Development Tools:** Python 3.12 with pip, Git, Android SDK Platform Tools (adb, fastboot)
- **Analysis Framework:** frida-tools (`pip install frida-tools`)
- **Platform Specific:** Xcode Command Line Tools (macOS only)
- **Mobile Devices:** Rooted Android or jailbroken iOS with frida-server installed

A.2.5 Benchmarks. FriddleBench contains two test applications:

- **iostest:** iOS Xcode project with custom AES-128 implementation, base64 encoding, and multiple operation modes (strcpy, base64, AES) to test different taint propagation scenarios
- **AndroidTest:** Android JNI project with both BoringSSL library integration and custom AES implementation, featuring multiple test modes and false positive test cases for complete taint analysis evaluation

A.3 Setup

A.3.1 Installation.

Test Applications Setup. Clone the Friddle repository: `git clone https://github.com/the7urn1ph34d/friddle.git`

iostest: Open `friddlebench/iostest/iostest.xcodeproj` in Xcode. Build and install to jailbroken iOS device.

AndroidTest: Navigate to `friddlebench/AndroidTest/`, build using Android Studio or `./gradlew assembleDebug`. Install APK to rooted Android device.

Frida Setup. Install frida-tools: `pip install frida-tools`
Root Android device or jailbreak iOS device.

Android frida-server setup: Download `frida-server-17.2.11-linux-arm64.xz` from https://github.com/frida/frida/releases/, extract the file, then push to device: `adb push frida-server /data/local/tmp/`. Run with root privileges.

iOS frida-server setup: Open Sileo, search for "frida" and install. frida-server will start automatically after installation.

Verify device connectivity: `frida-ps -U` or `frida-ps -H <ip>` for network connection.

A.4 Evaluation workflow

- **(C1):** Friddle achieves perfect correctness in tracking information flows across both Android and iOS native code for all test scenarios. This includes plaintext handling, Base64 encoding, custom AES encryption, and system library cryptographic operations. Testing on FriddleBench shows 100% true positive detection and correct false positive avoidance.
- **(C2):** Friddle provides instruction level dynamic taint analysis with measurable performance overhead (9,700x-65,800x slowdown). The overhead results from DBI instrumentation and JavaScript callout costs, but is acceptable for security analysis use cases requiring detailed taint tracking.

A.4.1 Experiments.

- **(E1):** [20 human-minutes]: Correctness Evaluation validates claims C1 by testing Friddle's taint tracking accuracy across test applications.
  Preparation: Deploy test applications (iostest for iOS, AndroidTest for Android) on both platforms.
  Execution:
  (1) Compile Friddle agent: `./compile_frida.sh ios.js` or `./compile_frida.sh android.js`
  (2) Launch test application on device
  (3) Inject Friddle: `frida -U -F -l ./ios_compiled.js` or `frida -U -F -l ./android_compiled.js`
  (4) Wait for console output showing successful initialization and execution:

```
TAINT_START_POS: [address]
TAINT_STOP_POS: [address]
processData at libandroidtest.so 0x7378c80000 0
     x7378d02b74
```

  (5) **True Positive Test:** Turn off false positive mode switch in application, input test data, check console output shows `output_buffer is fully tainted`
  (6) **False Positive Test:** Turn on false positive mode switch in application, input test data, check console output shows `output_buffer is not tainted`

Results: Compare detected flows against ground truth annotations. Expected results: 100% accuracy for both true positive and false positive detection.

- **(E2):** [20 human-minutes]: Performance Evaluation validates claims C2 by measuring Friddle's runtime overhead.
  Preparation: Install performance benchmark applications on test devices. Configure baseline measurements without instrumentation.
  Execution:
  (1) Run baseline test without taint analysis to measure native execution time
  (2) Launch test application on device
  (3) Inject Friddle agent: `frida -U -F -l ./ios_compiled.js` or `frida -U -F -l ./android_compiled.js`
  (4) Input test data and record execution time (application will toast the elapsed time)
  (5) Calculate overhead: (Time with Friddle) - (Baseline time without taint analysis)
  Results: Calculate overhead ratios. Expected results: 9,700x-65,800x slowdown depending on application characteristics.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.