Operation Mango: Scalable Discovery of Taint-Style Vulnerabilities in Binary Firmware Services

Wil Gibbs*, Arvind S Raj*, Jayakrishna Menon Vadayath*, Hui Jun Tay*, Justin Miller*, Akshay Ajayan*
Zion Leonahenahe Basque*, Audrey Dutcher*, Fangzhou Dong*, Xavier Maso†,
Giovanni Vigna‡, Christopher Kruegel‡, Adam Doupé*, Yan Shoshitaishvili*, Ruoyu Wang*

*Arizona State University
{wfgibbs,arvindsraj,jvadayat,htay2,jmill,aajayan,zbasque,dutcher,fdong12,doupe,yans,fishw}@asu.edu
‡University of California, Santa Barbara
{vigna, chris}@cs.ucsb.edu
†contact@xaviermaso.com

Abstract

The rise of IoT (Internet of Things) devices has created a system of convenience, which allows users to control and automate almost everything in their homes. But this increase in convenience comes with increased security risks to the users of IoT devices, partially because IoT firmware is frequently complex, feature-rich, and very vulnerable. Existing solutions for automatically finding taint-style vulnerabilities significantly reduce the number of binaries analyzed to achieve scalability. However, we show that this trade-off results in missing significant numbers of vulnerabilities.

In this paper, we propose a new direction: scaling static analysis of firmware binaries so that *all* binaries can be analyzed for command injection or buffer overflows. To achieve this, we developed MANGODFA, a novel binary data-flow analysis leveraging value analysis and data dependency analysis on binary code. Through key algorithmic optimizations in MANGODFA, our prototype Mango achieves fast analysis without sacrificing precision. On the same dataset used in prior work, Mango analyzed 27× more binaries in a comparable amount of time to the state-of-the-art in Linux-based user-space firmware taint-analysis SaTC. Mango achieved an average per-binary analysis time of 8 minutes compared to 6.56 hours for SaTC. In addition, Mango finds 56 real vulnerabilities that SaTC does not find in a set of seven firmware. We also performed an ablation study demonstrating the performance gains in Mango come from key algorithmic improvements.

1 Introduction

The Internet of Things has spawned a device ecosystem that allows users to control and automate everything in their homes—from their locks to their lights. This increase in convenience comes with increased security risks, as IoT devices frequently run (often out-of-date) firmware with vulnerable services [17, 26].

Modern IoT firmware frequently takes the form of an entire embedded Linux distribution that uses user-space binaries to expose services that can be accessed over the network. These services vary from web interfaces for device configuration, complex IoT-specific protocols (e.g., Matter [13] and Zigbee [14]), or custom file-sharing servers with varying levels of complexity. Many services invoke backend binaries to gather system and environment information, commit configuration changes, or render webpages.

While complex and feature-rich services make modern IoT devices powerful, this complexity breeds vulnerabilities [3, 39]. Therefore, automated techniques for scalable and precise detection of these vulnerabilities in the firmware of IoT are in pressing need.

Researchers have proposed analysis techniques to automatically identify vulnerabilities in IoT firmware, which generally fall into two categories: (1) static analysis techniques that attempt to statically identify vulnerabilities in the binaries in a firmware image using binary analysis techniques [9,36], and (2) dynamic analysis techniques that attempt to identify vulnerabilities through execution of said binaries [7, 22, 50, 51]. Broadly speaking, static approaches tend to discover more vulnerabilities (by covering much of the binary's behavior during analysis) while also reporting false positives (because static analyses cause over-approximation of the binary's behavior). At the same time, dynamic approaches lead to much fewer false positives (as dynamic techniques typically have a concrete input that triggers the vulnerability) while also missing vulnerabilities (due to incomplete code coverage).

Dynamic techniques are particularly challenging to apply to the domain of binary firmware services. Executing a target service is difficult due to dependencies on the underlying IoT operating system (OS) and hardware. Much of the research direction in this area is geared toward running a target service or emulating the hardware. But even the best firmware rehosting technique only successfully rehosts around 32% of 1,764 real-world firmware targets [44], hampering the application of dynamic techniques and making static techniques more promising.

The tantalizing promise of static analysis techniques is that they can be applied to *any* binary firmware service, as there

is no need to run or emulate the underlying OS or hardware. These techniques, while capable of finding real vulnerabilities, take significant resources: On Karonte's dataset of 49 firmware images SaTC [9] spends 870 hours processing, while Karonte [36] takes 463 hours. Even more troubling, to achieve even this level of scalability, Karonte and SaTC paid a significant price in either low coverage or high false negatives, respectively.

To improve scalability, Karonte introduced the concept of *Binary Dependence Graph* (BDG): a graph of nodes representing user-space binaries in a firmware image, with edges between those binaries denoting data flows. The BDG is anchored by binaries that are deemed to be network services, and only binaries that exchange data with them are analyzed. On Karonte's dataset, this reduces the number of binaries analyzed per firmware sample by Karonte to an average of 5 binaries and SaTC, which uses a similar strategy, to an average of 3, thus reducing the analysis requirement significantly from an average of 174 total binaries per firmware.

This technique, though novel, comes with a cost: the strict filtering induced by BDG, which is necessary to ensure scalability, filters out *actual vulnerabilities*. For example, the cross-binary vulnerability described in Section 2.2 is undetectable by the BDG technique, as implemented by Karonte and SaTC, as it requires limiting the analysis scope to a few binaries determined by their likelihood of interacting with user-input.

In this paper, we propose Mango, a static analysis technique for finding taint-style vulnerabilities in Linux-based user-space firmware services that scales so well that it removes the need for aggressive BDG-based filtering while maintaining reasonable precision. Mango's novel, context-sensitive static data-flow analysis, MANGODFA, which leverages intuitive execution patterns and trace shortening, requires no access to source code. MANGODFA uses both value analysis and data dependency analysis to enable precise reasoning about the *composability* and *exploitability* of values that flow into vulnerable sinks. For example, MANGODFA can correctly deduce that

```
sprintf(buf, "ls %s", itoa(input));
system(buf);
```

is not a vulnerability while

```
snprintf(buf, 50, "echo hello %s", input);
system(buf);
```

may be vulnerable to command injection.

To improve scalability, we draw inspiration from how human reverse engineers find taint-style vulnerabilities in software: Starting from vulnerability sinks and iteratively tracing backward through potential parent functions to determine if any data flows are likely vulnerable. Moreover, instead of analyzing each callee in every function along the call tree, human reverse engineers decide on a per-call basis if the call to a callee function would impact the data flow. These two steps are critical for reducing false positive rates and increasing analysis speed.

Using this inspiration, we implement two novel optimizations in MANGODFA, namely *Sink-to-Source Analysis* and *Assumed Nonimpact*, which mimic humans' actions during manual vulnerability discovery, and these optimizations are key for achieving scalability.

MANGODFA allows Mango to significantly increase the analysis scope *while simultaneously improving overall performance.* Mango analyzed *all 6,920 binaries* across the 49 firmware samples in the Karonte dataset in 946 hours, while SaTC analyzed 131 total binaries in 860 hours of CPU time and Karonte analyzed 153 binaries in 463 hours. While these performance numbers were done on different hardware at different times, Mango analyzed 27$\times$ more binaries in a comparable amount of time that it took SaTC to analyze 131 binaries. Mango's average per-binary analysis time is 8 minutes compared to 6.56 hours for SaTC and 3.0 hours for Karonte.

The current implementation of Mango supports two classes of vulnerabilities: CWE-78 (OS Command Injections) and CWE-121 (stack-based buffer overflows). Both comparative solutions, Karonte and SaTC, work explicitly on Linux-based user-space binary firmware services and not on monolithic or RTOS-based firmware. Through a comparative study on the firmware dataset on which SaTC and Karonte evaluated (with 49 unique firmware samples), we demonstrate that Mango generates 56 true positives that are never alerted by SaTC and similarly 203 true positives that are missed by Karonte.

We also conduct an ablation study to demonstrate the effect of Sink-to-Source Analysis and Assumed Nonimpact on efficiency and effectiveness. The improved efficiency of Mango allows us to conduct a large-scale vulnerability discovery on a new dataset with 1,698 firmware samples from nine vendors. Mango reports every potential bug it finds, 83,644 alerts in total. However, this number of alerts cannot be reasonably triaged. To aid in separating bugs from vulnerabilities, we introduce the concept of ranked alerts, which we call TruPoCs. Manual analysis on a sample set of these alerts suggests a true positive rate of 57% for command injections and 38% for buffer overflows. For every 10 command injection alerts a human analyst triages, 5 will be actual vulnerabilities.

Contributions. Our paper makes the following contributions:

- We build a novel data-flow analysis, MANGODFA, on the domain of integers and fixed-length strings, enabling scalable discovery of taint-style vulnerabilities. This new domain allows precise reasoning of string values that flow into vulnerable sinks, and is more precise than traditional taint analysis.
- We further propose two optimizations, Sink-to-Source Analysis and Assumed Nonimpact, which are critical for the efficiency and false negative reduction of MANGODFA.

- We implement MANGODFA in a prototype system, Mango, and evaluate it against the state-of-the-art solutions for finding taint-style vulnerabilities in firmware binaries.

In the spirit of open science, the source code and artifacts of Mango, along with a Docker image for replicating our experiments, can be found at https://github.com/sefcom/operation-mango-public. The contributions in this paper will also be integrated into angr [47].

2 Background

Before diving into the technical details of Mango, we provide the necessary background information on vulnerabilities in binary firmware services and discuss the associated challenges in discovering them with state-of-the-art static analysis techniques.

2.1 Firmware Types

Qasem et al. [34] categorized firmware into three types: Type I, where there is a clear separation between the Operating System (OS) and the contained firmware services (in terms of files and processes), Type II, where the OS and services are combined into a monolithic binary blob, and Type III, where there is no OS but only logic embedded in a blob.

Type II and Type III firmware are typically used to implement software for programmable logic controllers (PLCs) and real-time, low-power embedded devices (generally with less complicated logic). Type I firmware is more commonly seen in feature-rich and complex IoT devices such as routers, webcams, or networked CCTVs, where the OS is usually embedded Linux and services are ELF binaries (of varying architectures). A 2014 study by Costin et al. [15] found that over 86% of firmware samples in a large dataset of 32,356 images were Type I firmware using embedded Linux as the OS. Similar to prior work [6, 29] in this area, our research focuses on finding taint-style vulnerabilities in the user-space services of Type I firmware.

2.2 Motivating Example

Listing 1 shows a vulnerability in the Netgear R6400 router, which is in the dataset for Karonte and the ablation dataset SaTC used. The vulnerability here is straightforward: a command injection based on the `iserver_passcode` front-end keyword. SaTC missed this vulnerability, and Karonte would have as well if it supported command injections. Both of these tools missed this vulnerability because the border binary analysis for both tools does not include `dlnad` as a potential target. For example, SaTC prioritizes binaries with more references to frontend keywords and limits border binaries to three per firmware image. Due to the simple functionality and small size of `dlnad`, it is excluded by the border binary selection algorithm in SaTC.

Listing 1: Decompiled code of the `httpd` and `dlnad` binary from the Netgear R6400. Mango can identify this vulnerability while Karonte and SaTC will not analyze `dlnad`.

```
// R6400 /usr/sbin/httpd
int __fastcall sub_6F800(char *a1, int a2)
{
  sub_18764(a1, "iserver_passcode", v11, 2048);
  strcpy(v10, v11);
  v4 = acosNvramConfig_set("iserver_remote_passcode", v10);
  acosNvramConfig_save(v4);
  system("killall -SIGUSR1 dlnad");
}

...

// R6400 /usr/sbin/dlnad
int main(int argc, const char **argv, const char
↪  **envp)
{
  v10 = acosNvramConfig_get("iserver_remote_passcode");
  sprintf(v11, "echo \"NoDeviceName\r\n%s\" >
↪  /tmp/shares/forked_daapd.remote", v10);
  system(v11);
}
```

2.3 Firmware Vulnerability Analysis

For finding taint-style vulnerabilities in firmware, prior work proposes two genres of techniques: (1) firmware rehosting and dynamic analysis and (2) static analysis.

Firmware rehosting and dynamic analysis. Given the limited computing capabilities of IoT devices, researchers attempt to run firmware services on commodity systems, usually using an emulator such as Qemu [1] to leverage dynamic analysis techniques such as fuzzing. While researchers have focused on making taint-style analysis more approachable [11, 31, 32], the performance of dynamic techniques is limited by the effectiveness of firmware re-hosting, the quality of input seeds, and efficient coverage exploration, making it difficult for dynamic techniques to find taint-style vulnerabilities with reasonable scalability.

Consider the motivating example in Listing 1: to use a dynamic analysis technique, an analyst must first identify the connection that `dlnad` has to `httpd` through NVRAM variables. It would be an unlikely target because it does not actively process user input like a web server would. Further, even if an analyst wanted to analyze `dlnad`, it requires a specific combination of NVRAM values and properly formatted user input for the variables to trigger.

Static analysis. Static analyses sacrifice precision for higher scalability. Analyzing binaries statically requires a deep un-

derstanding of the binary format, the underlying architecture, and the operating system for which the binary is built. A typical static analysis workflow includes disassembling, lifting to an intermediate language (IL), CFG recovery, and various types of advanced analysis atop.

Symbolic execution on binaries [23, 47] allows tracking data flows in a binary: It simulates the execution of the target binary on a value domain of both concrete values and symbolic values (e.g., variables with unknown values), collects path constraints, and solves path constraints to derive possible values for symbolic values. The caveat is that symbolic execution is computationally expensive and suffers from path explosion [25]. Another technique for tracking data flows in a binary is static taint analysis [38]. It marks the source of the data and tracks it through the program, approximating the effects of function calls until reaching sinks. Static taint analysis may suffer from over-tainting, marking values as tainted when they are not, and under-tainting, missing tainted values.

Depending on the analysis goals, these static analyses may adopt value domains (e.g., the concrete domain where only concrete values are considered, or the value-set domain [4] where every value is modeled using its bits, lower- and upper-bounds, and its stride) with varied precision.

Firmware taint analysis. State-of-the-art static analysis techniques for finding firmware vulnerabilities, such as Karonte [36] and SaTC [9], start their analysis from *border binaries*, binaries that they determine to be reachable from the local network. SaTC takes it one step further: It identifies common keywords in web API interfaces and ignores all other interfaces.

While sometimes users knowingly communicate with their IoT devices via front-end web services (e.g., logging into a router's management portal), this is not the only available input vector for firmware services. Many services, such as `miniupnpd`, communicate without frontend interfaces but are open to receiving data from the local network, if not the Internet. By only focusing on border binaries and common web-API-style endpoints, Karonte and SaTC are likely to miss a large portion of vulnerability sources, which leads to false negatives (i.e., missed vulnerabilities).

While static analysis is the more viable approach for finding taint-style vulnerabilities, both Karonte and SaTC severely limit the potential vulnerability sources (by only analyzing border binaries and input locations with specific keywords), which impacts their ability to find vulnerabilities. Because MANGODFA is more scalable, Mango takes a different approach: It analyzes *every valid binary* in each firmware sample, flagging and filtering potential vulnerabilities.

2.4 Threat Model and Scope

Unlike Karonte or SaTC, we do not limit which binaries handle user input. Both use the concept of Border Binaries to limit the scope of their analysis to both reduce false positives and analysis times. However, Mango does not need this limit as it is able to analyze entire firmware samples in a reasonable amount of time. We do this exclusively on Linux-based user-space firmware binaries.

Bug reports and triggerable vulnerabilities. Because Mango runs on every binary in a given firmware, it identifies legitimate bugs in the current firmware regardless of whether the bugs are triggerable. For example, Mango can identify a vulnerable function that is unused or in an executable on the file system that an attacker cannot currently execute. While finding these bugs is essential, in this paper, we give a higher priority to finding *triggerable* vulnerabilities [1]. We coin these triggerable vulnerabilities *True PoCable vulnerabilities*, or *TruPoC*s in short. We implement a filtering system (described in Section 8) to prioritize bug reports that are more likely to be reachable by an attacker, thereby separating triggerable vulnerabilities from ordinary bug reports.

3 Challenges

It is possible to extend the functionality of Karonte and SaTC to analyze all binaries and all input sources in firmware samples. However, this will result in severe precision and scalability problems, which we demonstrate in an ablation study in Section 11. Here, we briefly present the major challenges that Mango faces and our solutions for solving each challenge.

3.1 Challenge 1: Precision

A static vulnerability discovery technique that reports a high number of false positives will quickly overwhelm users and fall out of use. Researchers find similar human behaviors when studying programmers facing compiler warnings [27].

Our solution to address the precision problem is three-fold. First, we realize that SaTC achieves high scalability but much lower precision by relying on a taint-tracking engine. SaTC can only track if an input value impacts key values at vulnerability sinks, but not *how* this input value is used. As such, some false alerts that SaTC generates result from imprecision during taint tracking, which frequently occurs when identifying command injection vulnerabilities as tainted values are often processed (and even sanitized) before reaching the sinks.

To address this problem, we create a novel data-flow analysis, MANGODFA, encompassing both value analysis and data dependency analysis over a mixed value domain of integers and fixed-length strings. MANGODFA tracks how a value is composed, e.g., which functions generated the value and

[1] When we conduct responsible disclosure for vulnerabilities to vendors, they would only accept vulnerability reports that come with Proof-of-Concept exploits (PoCs) that trigger the vulnerability and proactively reject vulnerability reports without PoCs.

which operations were applied to it before it is used at vulnerability sinks. We will present the details of MANGODFA in Section 5.

Second, as detailed in Section 5.3, we created function summaries for common library functions, many of which are in `glibc`. These function summaries are particularly helpful in string-related operations where functions have complex internal logic. Precisely emulating the binary code of library functions is expensive, and, as such, state-of-the-art tools fail to accurately capture the effects of these library functions.

Third, we implemented an alert filtering system that prioritizes alerts which are more likely to be reachable by an attacker that we call TruPoCs. Our goal is to provide TruPoCs for which users can easily generate PoCs and demonstrate the vulnerabilities. The details of how the ranking of alerts is determined can be found in Section 8.

3.2 Challenge 2: Scalability

To analyze every binary in each firmware sample, the analysis must be fast. Karonte takes an average of 9 hours to analyze a firmware sample where each sample only has five border binaries on average [36]; SaTC takes an average of 17 hours to analyze a firmware sample, but it hyper-focuses on a limited number of input sources for only a small set of binaries with keywords detected. The time SaTC needs for analysis increases significantly if we cover more than the initial border binaries. We extended SaTC to analyze all binaries in a firmware image (instead of the maximum of three border binaries). In our experiment using the Karonte Dataset, where each firmware image contains 127 analyzable binaries on average, SaTC took an average of six hours per binary and 762 hours per firmware image.

We propose two key optimizations in MANGODFA to significantly improve the scalability of static data-flow analysis.

The first optimization is *Sink-to-Source Analysis*, traces *backwards* from the callsite of a vulnerable sink to all potential sources of input that do not resolve to constant or known legitimate values. A key performance improvement is achieved through quickly eliminating paths that would otherwise resolve to constant values. We discuss the details of Sink-to-Source Analysis in Section 6.

The second optimization is *Assumed Nonimpact*, a technique for determining, at each call site, whether the callee function is essential to the data flow that leads to the vulnerability sink. This analysis technique can increase false positives caused by pointer aliasing and un-tracked global values; however, our evaluation shows a false-positive rate similar to SaTC. We present Assumed Nonimpact in Section 7.

4 Approach Overview

Mango takes a firmware image as input, analyzes all binary executables (not including library files) in the unpacked file system, and generates TruPoCs for potential vulnerabilities given the specified vulnerability class. Similar to Karonte and SaTC, Mango also supports multi-binary interactions and values derived from front-end keywords. After ingesting the firmware image, Mango automatically performs subsequent analyses. Finally, Mango outputs ranked POCs termed TruPoCs for the user to review.

The major analysis steps Mango performs are as follows:

Firmware Pre-processing. The main goal of Mango is to analyze ELF executables and identify potential vulnerabilities. It is important to note that we specifically do not analyze libraries as their vulnerabilities are more difficult to definitively categorize as true positives or false positives in a given firmware. Mango takes a firmware image as input, uses existing tools (such as binwalk [2]) to unpack the firmware sample, and finds all ELF executables. For all binaries, we gather their imported library function names. As long as the binary is not stripped and statically linked, we can recover function names such as `system` or `strcpy`. Later, we will use these symbols to determine whether or not to analyze a given binary.

Keyword Discovery. SaTC introduced keyword-guided analysis, which increases the attack surface user input can reach and reduces the number of binaries that need to be analyzed. Mango also gathers front-end keywords from HTML, JS, ASP, and PHP files. These keywords are a combination of string-literals and object fields that identify user input locations in backend binaries.

IPC Discovery. Mango uses Inter-Process Communication (IPC) data-flow tracking, first proposed by Karonte, to detect potential interactions between binaries in a given firmware sample. Specifically, Mango considers inter-process interactions that involve NVRAM entries and environment variables. Mango performs static analysis (based on MANGODFA) to collect all NVRAM and environment variables that each binary either sets or retrieves. Mango transforms these accesses into a dictionary of setters and getters keyed by the function and recovered value (if applicable) along with which binary the value originated from and its callsite within the binary.

Resolving Vulnerability Sinks. Resolving vulnerability sinks is the core step of Mango, where, for each vulnerability sink (e.g., the glibc `system` function) in each binary, it recursively resolves the sink to a *Rich Expression*, which captures the source values and all operations (e.g., `strcpy` and `sprintf` with format strings) applied to these source values. The sink-resolving procedure is inter-function within the same binary and inter-binary if source values originate from NVRAM entries or environment variables that Mango recovered during the IPC Discovery step.

Mango continues sink resolving until (1) it reaches the entry point of the binary, (2) all sink values are resolved to either *legitimate values* or *unresolvable values*, or (3) it times out. Legitimate values are not attacker-controllable or exploitable, such as constant values, constant pointers, or sanitized user

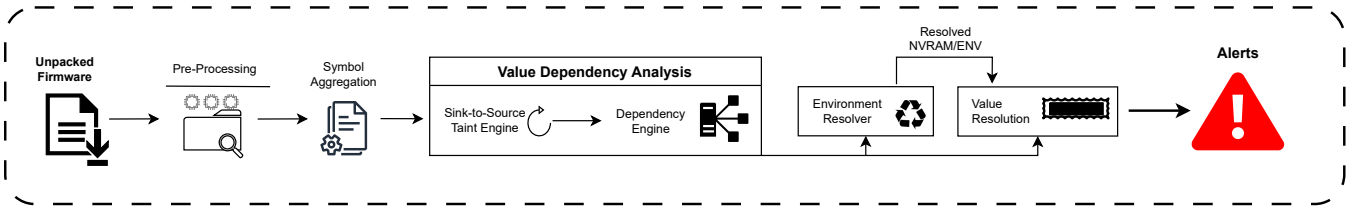

Figure 1: The workflow of Mango. Mango searches for all potential vulnerable and environment ELFs. It first analyzes all ELF files that can affect the environment and then analyzes all ELF files containing designated sinks with the context provided by the initial environment analysis.

input. All other values are unresolvable, including user or external input and buffers with unresolvable content due to missed or unsupported library calls.

5 MANGODFA

The core static data-flow analysis is MANGODFA, which is a combination of a static value analysis that tracks all possible values for each variable (in binary analysis, variables refer to value containers, such as registers, memory cells, etc.) at each program location and a data dependency analysis that tracks where each value is defined and used. This is a critical difference from prior work: Both Karonte and SaTC use taint-tracking, which means they can only mark certain variables and values as *tainted* by input data or not. In comparison, MANGODFA creates def-use graphs, with which Mango can reason about how any value was created from its source values and what operations have been applied. We call these expressions *Rich Expressions* because they capture not only the concrete values but also rich information about their history and dependencies.

An example Rich Expression is: `"df -d " .. itoa(atoi(input_1)) .. " -h " .. input_0)` where ".." represents the string concatenation operation. In this case, the underlying Rich Expression can be broken into its 4 components, `"df -d "`, `itoa(atoi(input_1))`, `" -h "`, and `input_0`, at any time with their corresponding dependencies already held in a global dependency graph. If a call to `system()` has a Rich Expression as the first argument (such as the one described), Mango can determine that `input_1` does not lead to arbitrary command injection as its value was derived from an integer. In contrast, `input_0` may lead to a command injection vulnerability as its origin value is unhindered. Neither Karonte nor SaTC could determine that `input_1` would not contribute to a vulnerability, which leads to higher false positive rates and reduced analysis speed due to over-tainting.

MANGODFA is the foundation for various sub-analyses on which Mango depends, such as calling convention inference and function prototype inference. Mango uses calling convention inference to determine the ABI of each function, which can be different even for functions within the same binary. Mango further uses function prototype inference to determine the number and the sizes of arguments for each function and if there is a return value. Both types of information are necessary for inter-function analysis. These sub-analyses are purely engineering work of existing research [5, 49] and are not our scientific contributions in this paper.

5.1 MANGODFA Basics

Value domain. MANGODFA employs a value domain that is specifically designed for finding command injection vulnerabilities. Basic value types in this domain include bit-vectors (integers and strings with fixed lengths) and a TOP ($\top$) value. Each value in this domain is either a *MultiValue* that contains at most N bit-vector values, a $\top$ value, or an *Expression* that is composed of *operations* that are applied on MultiValues. Each instance of MultiValue and Expression is of a fixed number of bits. MANGODFA is flow-sensitive but not path-sensitive, which means that whenever multiple paths merge, the values in each variable will be merged, creating MultiValues with more than one basic value inside. A MultiValue with more than N bit-vectors becomes a $\top$, and overly deep expressions (we empirically limit it to 20) also become $\top$. We choose a small N ($N = 5$) to guarantee fast convergence of the data-flow analysis.

Data dependency tracking. The data dependency analysis in MANGODFA is similar to traditional, source-code-based data dependency analysis: MANGODFA tracks where each variable is first defined and keeps a separate record for program locations where each defined variable is used. The definition record is associated with each value (bit-vectors or $\top$ values). During path merging, MANGODFA will adjust the association of definition records and re-associate definition records of old values to newly created $\top$ values if needed.

Memory model. The abstract state used in MANGODFA models "loads from" and "stores into" for registers, global memory regions, stack regions, and heap locations in a bit-precise manner. Because MANGODFA performs intra-function analysis by default, it only models one stack frame (also known as activation record).

5.2 Inter-function Analysis

To resolve sinks, Mango requires inter-function analysis. Building an inter-function MANGODFA from an intra-function MANGODFA is straightforward. At every call site, Mango makes a copy of the abstract analysis state thus far, creates a new stack frame to simulate the call, invokes a new MANGODFA on the copy of the abstract state, and then waits until the new MANGODFA instance terminates. Then, Mango merges all exit states that the new MANGODFA yields to get a new output state and uses that state to resume the outer-level MANGODFA analysis.

5.3 Function Summaries

The more accurately MANGODFA models the execution of a binary, the higher level of precision Mango can achieve. Karonte and SaTC are based on taint tracking, which models dependency relationships between input arguments and output values of certain library functions. Precisely propagating taints through many library functions (especially in `glibc`) can be difficult due to the complexity of the logic and implementations of these library functions, which may lead to excessive amounts of false positives and false negatives. Therefore, Mango implements function summaries for common library functions to ensure that values and data dependencies are modeled as accurately as possible.

Re-implementing library functions has the added benefit of reducing false positives caused by changes in value types. It is a common and normal execution pattern for programs to manipulate strings into non-injectable forms, such as using `atoi` to transfer the string into an integer or the `"%d"` format specifier for the `printf` function family, making injections impossible. SaTC does not recognize these input type changes and will report a (false-positive) vulnerability even if the tainted value is not injectable.

6 Sink-to-Source Analysis

Finding all input sources and performing forward data-flow analysis to find all vulnerability sinks is expensive. It is frequently unknown if values that depend on input sources will eventually flow into the vulnerability sink with a meaningful degree of control or exploitability. Shorter call traces for forward data-flow analysis imply a shorter analysis time because no extra time is spent analyzing irrelevant functions.

We propose a new workflow, Sink-to-Source Analysis, inspired by human reverse engineers. Reverse engineers tend to work *backwards* from potentially vulnerable program locations (e.g., a call to `system()`) to understand which input sources (if any) can reach this location instead of working from the input sources and exploring every possible path the inputs can take. These special program locations frequently coincide with vulnerability sinks for taint-style vulnerabilities.

Algorithm 1: Reverse Trace Analysis

```
1:  function ReverseTraceAnalysis(sink, max_depth)
2:      all_traces = { }
3:      traces = FindTraces(sink, max_depth)
4:      for trace in traces do
5:          state = { }
6:          white_list = list()
7:          rev_trace = {sink}
8:          for caller, callee in Reversed(trace) do
9:              intra_functions = CoarseAnalysis(caller, callee)
10:             for function in intra_functions do
11:                 if DependsOn(callee, function) then
12:                     rev_trace = rev_trace ∪ caller
13:                     white_list.append(function)
14:                 end
15:             end
16:             if DependsOn(callee, caller) then
17:                 rev_trace = rev_trace ∪ caller
18:             end
19:             else
20:                 if CalleeArgsResolved(callee) then
21:                     state = callee
22:                 end
23:                 all_traces[rev_trace] = (state, white_list)
24:                 break
25:             end
26:         end
27:     end
28: end
```

In the context of human reverse engineering, these special locations are also referred to as *beacons* [46].

Tracing backward from vulnerability sinks instead of forward to the sinks limits the explored program paths to only those directly relevant to the sinks. By analyzing caller function by caller function, backward through a call stack, Sink-to-Source Analysis limits the analysis required to determine reachability. Source-to-Sink Analysis may explore any number of unimportant paths to discover how or even if tainted data will reach the sink.

DTaint [10] employs a similar Sink-to-Source approach, but it requires analysis of every function in the callgraph backwards to the highest-level source, thereby analyzing many functions that ultimately do not affect the sink. During our Sink-to-Source analysis, if MANGODFA determines that a data flow flows into an unexploitable variable (e.g., a properly sanitized string or a string that is ultimately converted into an integer), it immediately abandons this variable and terminates the data flow. Consider the example `sprintf(buf, "cal %d", year)`, MANGODFA performs a data-dependency analysis that determines that `buf` will never contain a vulnerable string and terminates its analysis. However, DTaint will continue to analyze up through the call chain until it can find a source for `year`.

```
void foo() {
    bar("Hello!");
}

void bar(char *unused) {
    char *name_buf = nvram_get("your_name");
    say_hello(name_buf);
}

void say_hello(char *buf) {
    char[256] cmd;
    sprintf(cmd, "echo \"Hello %s\"", buf);
    system(cmd);
}
```

Listing 2: An example of Sink-to-Source Analysis.

Our Sink-to-Source Analysis algorithm, as shown in Algorithm 1, begins by finding all possible traces to the sink for a given CFG. We perform a coarse-grain value dependency analysis on each caller-to-callee pair, working backward from our sink in every trace. The analysis results determine if data reaching the sink solely depends on data from the specified caller in the trace.

Suppose there is a call to `system` with a command that is entirely dependent on the result of reading a file, environment variables, NVRAM variables, or data from the network; the analysis is halted as higher-level caller functions in the call-stack will not contribute to the data that `system()` ultimately depends on. If no variables are dependent on the caller function arguments, there is no need to analyze anything further, but we must keep the current trace depth for data dependencies inside the caller function. If the data depends on function arguments, the analysis continues up the call stack. However, if during the analysis of the next call-depth, all callee functions have constant or fully resolvable values, then we discard the current call-depth and retain a final state with set arguments for the callee. To analyze the caller function of a fully analyzed function, we simply repeat all steps from the beginning except instead of the caller/callee pair being a parent function and the sink, the parent function of the sink becomes the new callee.

In the case of Listing 2, MANGODFA starts analyzing from *say_hello* and marks *buf* with an unresolved value as it could contain anything at this point in the analysis as far as MANGODFA is aware. MANGODFA will trace the dependency of *buf* to *say_hello*'s arguments and decide to analyze *say_hello*'s parent functions to resolve the values in *buf*. The analysis will continue from *bar*, the parent function of *say_hello*, and MANGODFA will determine that there is no need to analyze any of *bar*'s parent functions, in this case *foo*, as the data in *buf* is only dependent upon the result of the call to *nvram_get*.

Algorithm 2: Assumed Nonimpact

```
1:  function AnalyzeFunction(state, white_list, sink)
2:      for func in white_list do
3:          intra_funcs = state.TaintAnalysis(func)
4:          if DependsOn(sink, func.ret_val) then
5:              if func.call_depth == maximum_depth then
6:                  continue
7:              end
8:              intra_white_list = {}
9:              for intra_func in intra_funcs do
10:                 if DependsOn(intra_func, func.ret_val) then
11:                     intra_list.append(intra_func)
12:                 end
13:             end
14:             AnalyzeFunction(func, intra_list, func.ret_val)
15:         end
16:     end
17: end
```

The ablation study described in Section 11.5 evaluates the benefits of Sink-to-Source Analysis. With all things equal, Sink-to-Source Analysis improves analysis time by 25.61%.

7 Assumed Nonimpact

Traditional inter-function data-flow analysis cannot effectively analyze firmware at scale. One fundamental problem is that forward inter-function data-flow analysis usually must enter and analyze every callee; otherwise, it misses the artifacts that the callee generates. However, consider well-formed C programs where functions all have clean input/output interfaces and are defined by their arguments and return variables: in this case, we may safely skip the analysis of certain callees if pointers to the values and variables that we care about are not passed into these functions. Skipping these callees means that we can consider them a black-box, and that we cannot capture any information inside them. This is acceptable because MANGODFA only cares about the effects that would impact the values that are being tracked.

Therefore, we propose *Assumed Nonimpact*, a best-effort execution strategy for function callees during data-flow analysis. Selectively analyzing callees significantly reduces the analysis time of MANGODFA.

Algorithm 2 describes the process of Assumed Nonimpact. When MANGODFA analyzes a new function *A*, it recovers calling convention, prototype, and accessed global variables for each callee that *A* calls. With calling conventions and prototypes of callees, MANGODFA has enough information to determine, at every call site, if (1) any pointer that points to values and variables that MANGODFA tracks is passed to the callee function, or (2) any values that the callee function returns and is ultimately used by the sink in some form. When neither situation holds, we assume the call will not impact the

values and variables that MANGODFA tracks and skip the call.

TChecker [28] discusses a similar technique to Assumed Nonimpact. The novelty in Mango lies in adapting the technique for binaries, whereas TChecker only works on PHP applications. Achieving this in binaries is significantly more challenging compared to PHP source code because high-level semantic information, such as data structures, are discarded during compilation. Additionally, analyzing binary code requires reasoning about binary-specific features such as calling conventions, function prototypes, and precisely tracking data flows. Further, Mango works on binaries of multiple architectures, each with nuances, further increasing the difficulty.

While MANGODFA caches the calling conventions and prototypes for callee functions, it must determine whether to skip a call at each call site. The decision to skip a callsite depends on the value of the arguments to the callee and if those values are mutable as MANGODFA progresses, making the decision heavily dependent on the context of the call.

Loss of soundness. The fundamental assumptions underpinning Assumed Nonimpact are that (1) all callees are well-formed (i.e., they will not add a constant to the stack pointer and use it to overwrite values that belong to the caller's stack frame) and (2) all recovered calling conventions, prototypes, and accesses to global variables are accurate. These assumptions do not always hold for stripped binaries, especially when structs (not pointers to structs) are in use, where MANGODFA frequently recovers incorrect calling conventions and prototypes. Another error source is global struct member accesses, where it is difficult to determine the size of global structures, leading to incorrect assumptions of what ranges of data a callee function may access. Our evaluation shows that Assumed Nonimpact works well in most cases, and we hope future advances in binary analysis research will alleviate these issues and reduce the loss of soundness.

8 From Alerts to TruPoCs

When disclosing a vulnerability to vendors, they often require a PoC to demonstrate that the vulnerability exists and is triggerable. As such, Mango focuses on finding TruPoCs. We design a filtering system in Mango that takes vulnerability alerts it finds and generates TruPoCs. These vulnerabilities have clearly defined paths through single or multiple programs stemming from user-controlled sources to sinks.

During static analysis described in Section 5, we build a dependency graph of data flows from all sources to sinks. We use this dependency graph to discover and categorize possible interesting sources of input.

We categorize potential external data sources into five categories: *file_ops* consists of file operations, such as `fgets`, `fread`, etc. *network_ops* covers `socket`. *env_ops* contains all references to `frontend_params`, `getenv` and `nvram_get` style operations, which retrieve stored data. *argv* denotes paths that appear to originate from a program's command line. Finally, *unknown* is a category used for all data from any sources not listed above.

In the context of firmware, it is easy to identify sources for *env_ops*. If the *env_op* of an alert has a known location that sets its value or it is determined to be a user-controllable `frontend_param`, then Mango will report the alert as a TruPoC.

9 Environment Resolution

A key advancement that Karonte suggested is modeling multi-binary interactions through analyzing Inter Process Communications (IPCs). During multi-binary data-flow analysis, Karonte propagates constraints from source binaries to sink binaries, including environment information extracted along the way. The environment resolution in Mango employs a similar tactic. However, Mango does not perform symbolic exploration or propagate constraints across binaries that interact through IPCs. Instead, Mango directly links setter values and getter variables during data-flow analysis, which achieves *cross-binary* vulnerability sink resolution. Our solution is more scalable than Karonte because it avoids path explosion issues in symbolic exploration and runtime overhead involved in constraint solving. Mango also applies the same environment resolution approach to front-end keywords analysis that SaTC pioneered.

10 Implementation

We implemented the Mango prototype in approximately 9,500 lines of Python code. The multi-binary analysis for finding NVRAM entries and environment variables as well as MANGODFA was built on top of angr [41], a widely used multi-architecture binary analysis framework. The core of Mango includes the IPC and front-end variable identification and the data-flow analysis on firmware binaries. Mango supports all architectures and platforms that angr supports, e.g., X86, AMD64, ARM, AArch64, MIPS, PowerPC, etc. We also implement a set of scripts to enable the parallelization of Mango analysis in local Docker containers or a Kubernetes cluster.

11 Evaluation

We evaluated Mango using firmware running on real-world routers to answer the following research questions:

RQ1. How effective and efficient is Mango compared to SaTC at identifying vulnerabilities on real-world firmware?
RQ2. How do the scalability optimizations Sink-to-Source Analysis and Assumed Nonimpact impact scalability and precision?

RQ3. How does Mango perform on a significantly large-scale dataset of firmware (100× larger than what was used in prior work)?

11.1 Datasets

We evaluate Mango on three different firmware datasets.

Karonte Dataset. The Karonte Dataset[2] contains 49 firmware samples across four different vendors (Netgear, D-Link, TP-Link, and Tenda). There are 3,599 binaries in all of these firmware samples.

SaTC Selected Dataset. SaTC also compared against Karonte on seven additional firmware images from Netgear, D-Link, and Tenda. As such, we will compare against SaTC on these seven additional images containing 384 binaries in total.

Large-scale Dataset. We use the dataset that Greenhouse uses [44]. This dataset comprises 1,698 firmware samples from nine different vendors: Netgear, Asus, Belkin, Linksys, TPlink, Trendnet, Tenda, D-Link, and ZyXEL containing over 385,000 binaries.

11.2 Methodology

For every dataset, we run Mango on each ELF binary in the firmware sample that contains a vulnerability sink. In this evaluation, we only compare against SaTC and not Karonte as SaTC strictly improves upon the amount of vulnerabilities found per firmware image compared to Karonte.

11.2.1 Vulnerability Confirmation

Where possible, we manually analyze a subset of Mango's TruPoCs to determine which are true positives (actual vulnerabilities) and which are false positives (not vulnerable). The TruPoCs are considered true positives if an attacker in our threat model (Section 2.4) can exploit the vulnerability as a user connected to the device with credentials. We have generated PoCs and successfully exploited 70 vulnerabilities based on our TruPoCs. We are working with vendors to responsibly disclose these vulnerabilities.

11.3 Karonte Dataset Results

Table 1 shows the results of Mango and SaTC on the Karonte Dataset. After analyzing 3,599 binaries over 946 hours, Mango generated 2,310 TruPoCs. This compares to SaTC analyzing 131 binaries in 860 hours and generated 144 TruPoCs.

Mango achieves comparative speeds to SaTC despite analyzing 6,920 binaries vs SaTC's analysis of 131 binaries. The speed of our analysis, despite the scope of our threat model, is attributed to our contributions of Assumed Non-impact and Sink-to-Source Analysis optimizations (demonstrated through an ablation study in Section 11.5).

To better understand Mango's effectiveness, we manually analyzed a subset of the generated TruPoCs. Table 4 shows the results of our manual analysis. We randomly sampled 100 Mango command injection TruPoCs and manually verified them. 57 were true positives, with a 57% true positive rate, 43 were false positives, with a 43% false positive rate. With the addition of TruPoCs, Mango's 57% true positive rate is higher than SaTC's at 32.77%.

We also randomly sampled 230 Mango buffer overflow TruPoCs and verified them. Mango found 109 and 121, giving it a true positive rate of 47%.

Our manual analysis of the buffer overflow results found several cases where Mango identified a vulnerability that Karonte or SaTC could not (due to their reliance on the Binary Dependence Graph). Our motivating example (Listing 1) showcases this exact issue.

11.3.1 Baseline Comparison

Taint-style vulnerability analysis is usually paired with the inherent worry of whether the analysis itself has any effect—with the level of vulnerabilities that we identify, would a naïve approach to generate a TruPoC on any vulnerable sink function call be just as effective? To answer this question, we designed a naïve baseline approach.

The baseline approach marks every vulnerable sink function call as a potential vulnerability. However, the user's effort does not stop there, as they must verify the vulnerabilities by analyzing the context of the vulnerable sink function call. Therefore, a fair comparison is the TruPoCs generated by Mango compared to the number of context paths backward from a vulnerable sink function call. We limit the depth of the backward context traversal to 7, mimicking how far back Mango will go to find unique path contexts.

Table 2 shows the results of this comparison for the command injection Mango TruPoCs and vulnerable context paths and the same for buffer overflows. Mango significantly reduces the burden on the human analyst by reducing the number of vulnerable context paths they need to reason about from 7,251,623 to 2,094 for buffer overflows and 347,405 to 216 for command injections.

11.4 SaTC Selected Dataset Results

SaTC evaluated their tool against 7 selected firmware, and on these firmware, Karonte produced zero true positives. We evaluated Mango against these same firmware images, and to do so fairly, we evaluated only on command injections just as SaTC did in their foundational work. Table 3 shows the results of this experiment.

[2] https://github.com/ucsb-seclab/karonte#dataset

		SaTC						Mango					
Vendor	**Samples**	**Total**	**Cmdi**	**Overflow**	**Total Time**	**Analyzed Bins**	**TruPoCed Bins**	**Total**	**Cmdi**	**Overflow**	**Total Time**	**Analyzed Bins**	**TruPoCed Bins**
Netgear	17	138	42	96	450:27	50	22	2,089	143	1,946	502:29h	1,950	127
D-Link	9	2	0	2	174:34	18	7	36	0	39	178:08h	696	18
TP-Link	16	0	0	0	153:55	42	17	56	0	56	192:32h	610	8
Tenda	7	4	2	2	82:02	21	5	129	73	56	73:11h	343	21
Total	49	144	44	100	860:58	131	52	2,310	216	2,094	946:22	3,599	174

Table 1: Comparison of SaTC and Mango on Karonte Dataset. *Analyzed Bins* are the amount of binaries Mango analyzed while *TruPoCed Bins* are the amount of binaries with TruPoCs. A subset of TruPoCs were manually analyzed for correctness, the results of which are shown in Table 4.

	Buffer Overflows	**Command Injections**
TruPoCs	2,094	216
Vulnerability Sinks	4,725	3913
Vulnerability Context Paths	7,251,623	347,405

Table 2: An analysis of Mango against a baseline approach. In the naïve baseline approach, every vulnerable sink function call is marked as a potential vulnerability. However, to verify the user must analyze the paths backwards from the vulnerable sink function. Therefore, Mango reduces the need to look at Vulnerable Context Paths to TruPoCs, thus reducing significant verification effort from the human analyst.

Mango analyzed 384 binaries in 49 hours and generated 145 TruPoCs, while SaTC analyzed 21 binaries in 93 hours and generated 65 TruPoCs. Mango generated a total of 89 (61%) true positive TruPoCs versus SaTC's 36 (55%). In addition to more than doubling SaTC's true positive TruPoCs, Mango also ran 90% faster than SaTC.

Mango generated 145 command inject TruPoCs from 31 binaries whereas SaTC's 65 originate from 13 binaries. There is an overlap of 13 binaries between Mango and SaTC for these seven selected firmware. Therefore, the TruPoCs generated by the other 18 binaries that Mango found are false negatives for SaTC. However, SaTC found 11 TruPoCs in the seven firmware for which Mango did not generate a TruPoC, resulting in a few false negatives for Mango. Despite this, the reduced analysis scope induced by the Binary Dependence Graph technique introduces significant false negatives.

11.5 Ablation Study

To answer RQ2, we evaluated Mango on the Netgear firmware R6400v2 and measured the impact of Assumed Nonimpact and Sink-to-Source Analysis. All runs were done on the exact same set of binaries, and the times reported are only on binaries that were ran successfully across all options. As a baseline, we ran Mango normally, enabling both Assumed Nonimpact and Sink-to-Source Analysis.

Table 5 shows the results of this analysis on the 60 binaries in the firmware for command injection sinks. Mango achieved an average execution speed of 4 minutes per binary and an overall run-time of 275 minutes.

11.5.1 Disabling Assumed Nonimpact

Theoretically, Assumed Nonimpact allows Mango to run faster by analyzing fewer paths and functions along the way to our sink. With only Assumed Nonimpact disabled, i.e., analyzing every path and function in-line to the sink, the runtime is 632 minutes with an average of 10 minutes spent on each binary. Thus, Assumed Nonimpact provided a speed increase of 129.82%. It should be noted that Assumed Nonimpact also generates more TruPoCs and true positives than Mango without Assumed Nonimpact.

11.5.2 Disabling Sink-to-Source Analysis

We conducted a similar analysis by disabling Sink-to-Source Analysis. This forward tracing approach results in a decrease in speed against our Sink-to-Source Analysis. The average runtime of a binary is 5 minutes with a total of 345 minutes. Enabling Sink-to-Source Analysis results in a speed increase of 25.61%. This is notably faster than default Mango and generates many more TruPoCs and true positives.

11.5.3 Disabling Both Optimizations

Finally, we evaluated our approach against the standard taint approach by disabling both Assumed Nonimpact and Sink-to-Source Analysis. Without these optimizations, the average time taken to analyze a binary was 26 minutes, and it took 1,575 minutes to analyze the given firmware.

The number of TruPoCs generated for this approach is much lower than our approach of using both Assumed Nonimpact and Sink-to-Source Analysis. To analyze all context sensitive paths in a timely manner, we place a timeout of six minutes on individual Sink-to-Source path analysis. There can be hundreds if not thousands of paths from all sinks that need to be analyzed (as Section 11.3.1 shows). We used Sink-to-Source Analysis to find the highest common call-depth for triggering bugs, which distills the shortest unique path to each sink.

11.6 Large-scale Dataset Results

The Large-scale Dataset consists of largely ARM and MIPS based router firmware in addition to some NAS firmware. We

Vendor	Device	SaTC					Mango				
		TruPoC	TP	Time	TruPoC Bins	Total Bins	TruPoC	TP	Time	TruPoC Bins	Total Bins
Netgear	R6400	4	4	30:23h	1	3	16	9	6:30h	4	76
Netgear	R7000	5	2	11:34h	1	3	23	14	8:50h	5	85
Netgear	XR300	10	4	22:57h	3	3	59	50	18:41h	9	65
D-Link	DIR878	22	16	8:22h	3	3	8	7	4:10h	4	40
Tenda	AC15	10	4	9:23h	2	3	16	3	1:20h	4	39
Tenda	AC18	10	4	8:25h	2	3	26	3	8:15h	4	39
Tenda	W20E	4	2	0:55h	1	3	3	3	1:35h	1	35
Total	-	65	36	93:20h	13	21	145	89	49:21h	31	389

Table 3: Comparison of SaTC and Mango on SaTC Selected Dataset. *TruPoC Bins* are the amount of binaries with actual TruPoCs found.

	TruPoCs	TP	TP Ratio	FP	FP Ratio
Command Injection	100	57	57%	43	43%
Buffer Overflow	230	109	47%	121	53%

Table 4: Results of manual analysis on a subset of Mango's TruPoCs generated from the Karonte Dataset.

performed the evaluation on a Kubernetes cluster composed of 2.30GHz Intel Xeon CPUs, with each process running strictly on one core and at least 5GB of RAM allocated per binary.

Table 6 shows the results, and Mango analyzed 770,374 binaries, generating 10,834 TruPoCs over a total CPU time of 339 Days 21h. On average, Mango only took 38.12s to fully analyze a binary and 5h 32m to analyze any individual firmware. This time per firmware is much lower than the results reported by either SaTC or Karonte.

12 Discussion

We discuss our view on the quality of the data set as well as the limitations of Mango in this section.

12.1 Threats to Validity

Based on our experience, there are many more taint-style vulnerabilities in IoT firmware than in traditional software. We believe this is partially due to the lack of proper software engineering practices in major IoT device vendors as well as the lack of security defenses in compilers and OSes for embedded devices. For example, many MIPS binaries that Mango analyzed did not have stack canary enabled, and the majority of MIPS CPUs do not support address space layout randomization. Therefore, while our proposed techniques and algorithms can apply to finding taint-style vulnerabilities in traditional software and commercial off-the-shelf (COTS) binaries, the improvement over traditional forward data-flow analysis may not apply there. However, given the sheer number of IoT devices in our community, Mango represents a step forward in automated firmware vulnerability analysis.

We used the Karonte Dataset for the sake of consistency as it is a common point that was evaluated on both SaTC and Karonte. However, the dataset is slightly outdated as there are many more newer firmware from all of the presented vendors. In an attempt to combat this and show Mango is practical even on newer and larger firmware we present an evaluation based on a larger and newer dataset as shown in Table 6.

12.2 Limitations

Assumed Nonimpact may cause soundness losses. As discussed in Section 7, Assumed Nonimpact may lead to soundness issues due to incorrect or unreliable analysis results from calling convention, function prototype, and function I/O interface inference.

Balancing between false positives and false negatives. The design of Mango inherently determines that it may report false positives (i.e., vulnerability alerts that are not real vulnerabilities). These can be caused by broken data dependencies (usually the result of incomplete CFGs or missing edges on a CFG) or intended functionality. Unlike Karonte and SaTC, which aim for low false positive rates, we believe that it is vital for a vulnerability detection tool to report as many TruPoCs as possible (i.e., lower false negatives) while keeping the false positive rate at an acceptable level. Many vulnerabilities that Mango finds are impossible for Karonte or SaTC to find.

12.3 Future Work

Although Mango has significantly improved finding vulnerabilities in firmware using static analysis, there is yet more room for improvement. Currently, Mango focuses solely on command injections from various functions and buffer overflows resulting from *strcpy*. The scope of covered vulnerabilities can be widened to path traversals, SQL injections, and other string-related vulnerabilities.

Firmware	Assumed Execution	Source-To-Sink	Average Per Binary (minutes)	Total (minutes)	TruPoCs	TP	Errors
R6400v2	✓	✓	4:35	275:08	16	9	13
	✗	✓	5:45	345:41	21	9	37
	✓	✗	10:32	632:18	13	5	22
	✗	✗	26:15	1,575:02	8	1	29

Table 5: Results of the ablation study where four different configurations of Mango are evaluated on R6400v2 firmware.

Vendor	Samples	Binaires				Time	AVG Time	AVG Bin Time
		Total	TruPoCs	Error	OOM			
Netgear	305	182,600	6,716	4,240	52	84 Days 3h	6h 37m	39.82s
Asus	158	104,422	635	1,174	45	46 Days 16h	7h 05m	38.62s
Belkin	62	20,018	2,102	353	12	14 Days 13h	5h 38m	62.94s
Linksys	67	46,470	211	440	8	20 Days 14h	7h 23m	38.33s
TPlink	484	239,020	166	3,339	93	89 Days 5h	4h 25m	32.26s
Trendnet	178	41,878	31	5,585	3	6 Days 18h	0h 51m	13.02s
Tenda	104	29,650	56	576	9	18 Days 6h	4h 12m	53.24s
D-Link	320	95,788	907	2,626	25	54 Days 1h	4h 12m	50.56s
ZyXEL	20	10,528	10	254	2	4 Days 0h	4h 48m	32.85s
Total	1,698	770,374	10,834	18,587	249	339 Days 21h	5h 32m	38.12s

Table 6: Large-scale evaluation run on 1,698 firmware samples. This table lists the vendors belonging to the firmware run on the largescale dataset for Mango's analysis and all numbers shown are relative to said vendor.

In addition, buffer overflows from bounded operations such as *strncpy*, *snprintf*, and *memcpy* can be introduced with some basic value set analysis. A more comprehensive application of value set analysis [4] allows for informed decisions on which branches should be taken to trigger a vulnerability which will improve Assumed Nonimpact's performance. This can also filter out false positive results with the bounding of buffers provided by value set analysis.

13 Related Work

Static solutions. Karonte [36] was one of the first to introduce multi-binary interactions to static analysis in the context of IoT devices. Their approach models several input sources, such as environment variables and NVRAM entries, and propagates that information from producer binaries to consumer binaries. SaTC [9] continues their approach, but instead of including network interactions as their sources, they use keywords found in front-end files such as JavaScript and PHP. They capture directly exploitable vulnerabilities via user input in these front-end files. This increases the likelihood that TruPoCs will be true positives. However, they ignore other critical binaries in favor of their border binaries. Both Karonte and SaTC severely limit the scope of analysis for efficiency and reduced false positive rates. For example, Karonte and SaTC only analyze binaries that are obviously reachable from the outside network; SaTC limits input sources even more by enforcing keyword matches. As such, they both suffer from high numbers of missed vulnerabilities, as is evidenced by the discrepancy of true positives between Mango and SaTC on Karonte's dataset. Mango avoids this limitation by significantly increasing the analysis efficiency through design benefits (using static data-flow analysis instead of taint tracking and symbolic exploration) and algorithmic improvements (Sink-to-Source Analysis and Assumed Nonimpact).

Saluki [20] produces an extremely fast static analysis through the micro execution of paths in the binary. Though this approach is much faster than previous work, it requires precise vulnerability descriptions using their custom language, making the tool difficult to use and highly specific. Mango takes a much more general approach where categories of sinks defined by function names are set. Arbiter [45] begins to cross into the boundary of dynamic analysis with their use of Under-Constrained Symbolic Execution (UCSE). They approach the problem of vulnerability verification with a static taint analysis followed by UCSE for reachability verification. However, their approach only works on x86-64 binaries, unlike Mango, which also runs on MIPS and ARM binaries. Firmalice [40] provides backdoor detection and verification in firmware binaries using symbolic execution but is bound by the overhead of symbolic exploration. Similarly, FIE [18] uses symbolic execution to analyze open-source MSP430 firmware binaries, but there are various firmware binaries that are intractable to analyze with this tool.

Firmware rehosting and dynamic analysis. Firmware rehosting refers to the process of running or emulating firmware on workstations or servers for interaction and security auditing. Most work in this area conducts full-system emulation of firmware targets, but several have extracted pieces to be

analyzed separately [15, 35]. Once rehosting succeeds, researchers usually move on to vulnerability testing (e.g., using Metasploit) or fuzzing [8, 37]. Costin et al. [16] used full-system emulation to rehost COTS firmware, and then used web penetration tools to find vulnerabilities in web servers on rehosted firmware. Firmadyne [7], Avatar [48], Avatar^2 [33], and FirmAE [22] provides more automation into the full-system rehosting process by QEMU-system to rehost firmware images in a large scale.

Unfortunately, rehosting is usually exceptionally tedious and error-prone, and access to peripherals only available on the original IoT devices is lost. Therefore, PROSPECT [21], CHARM [43], and SURROGATES [24] conducts hardware-in-the-loop operations during firmware emulation. HALucinator [12] and DICE [30] emulate Hardware Abstraction Layers (HALs) and the DMA (Direct Memory Access) channel. Usually, fuzzing is a natural application after firmware rehosting [19, 42, 50, 51].

14 Conclusion

IoT devices are riddled with security vulnerabilities, not only due to shoddy coding practice but also without many of the layers of modern defenses that we take for granted on desktops and mobile devices. And yet, we invite these devices into our homes and onto our networks, where attackers leverage their (apparently) innate insecurity to gain a foothold and wreak havoc.

We believe that Mango represents a significant step forward in identifying these vulnerabilities *before* an attacker ruins our day. By quickly statically analyzing all binaries on a firmware for security vulnerabilities, we can help raise the bar for IoT device security.

Acknowledgement

This research project has received funding from the following sources: Defense Advanced Research Projects Agency (DARPA) Contracts No. HR001118C0060, FA875019C0003, N6600120C4020, and N6600122C4026; the Department of the Interior Grant No. D22AP00145-00; the Department of Defense Grant No. H98230-23-C-0270; the Advanced Research Projects Agency for Health (ARPA-H) Grant No. SP4701-23-C-0074; and National Science Foundation (NSF) Awards No. 2146568, 2232915, and 2247954.

References

[1] "Qemu," 2023, https://www.qemu.org.

[2] "Refirmlabs/binwalk: Firmware analysis tool," 2023, https://github.com/ReFirmLabs/binwalk.

[3] Balaji, "A zero-day vulnerability in tp-link router let hackers gain admin privilege and take full control of it remotely," 2019, https://gbhackers.com/tp-link-router/.

[4] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *International conference on compiler construction*. Springer, 2005, pp. 250–254.

[5] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[6] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1687–1704.

[7] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *NDSS*, vol. 1, 2016.

[8] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[9] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems." in *USENIX Security Symposium*, 2021, pp. 303–319.

[10] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 430–441.

[11] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 196–206.

[12] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 1201–1218.

[13] Connectivity Standards Alliance, "Build with Matter | smart home device solution – CSA-IOT," https://csa-iot.org/all-solutions/matter/.

[14] ——, "Zigbee | complete IOT solution – CSA-IOT," https://csa-iot.org/all-solutions/zigbee/.

[15] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 95–110.

[16] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 437–448.

[17] CyberTalk, "FBI issues warning, unpatched and outdated IoT," 2023, https://www.cybertalk.org/fbi-issues-warning-unpatched-outdated-iot/.

[18] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution." in *USENIX Security Symposium*, 2013, pp. 463–478.

[19] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[20] I. Gotovchits, R. Van Tonder, and D. Brumley, "Saluki: finding taint-style vulnerabilities with static property checking," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, vol. 2018, 2018.

[21] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: peripheral proxying supported embedded code testing," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2014.

[22] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual computer security applications conference*, 2020, pp. 733–745.

[23] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[24] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[25] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the path explosion problem in symbolic execution-driven test generation for programs," in *2010 19th IEEE Asian Test Symposium*. IEEE, 2010, pp. 59–64.

[26] Kudelski IoT, "Why is it so hard to keep IoT devices up to date and secure?" 2023, https://www.kudelski-iot.com/insights/why-is-it-so-hard-to-keep-iot-devices-up-to-date-and-secure.

[27] G. Kudrjavets, A. Kumar, N. Nagappan, and A. Rastogi, "The unexplored terrain of compiler warnings," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 283–284.

[28] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2175–2188.

[29] A. Mandal, P. Ferrara, Y. Khlyebnikov, A. Cortesi, and F. Spoto, "Cross-program taint analysis for iot systems," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1944–1952.

[30] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of dma input channels for dynamic firmware analysis," in *IEEE Symposium on Security and Privacy (SP)*, 2021.

[31] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "Straighttaint: Decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 308–319.

[32] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "Taintpipe: Pipelined symbolic taint analysis," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 65–80.

[33] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.

[34] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–42, 2021.

[35] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: On the security of bootloaders in mobile devices." in *USENIX Security Symposium*, 2017, pp. 781–798.

[36] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1544–1561.

[37] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 19–36.

[38] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 74–83.

[39] SC Staff, "Netcomm, TP-Link routers impacted by critical bugs," 2023, https://www.scmagazine.com/brief/device-security/netcomm-tp-link-routers-impacted-by-critical-bugs.

[40] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[41] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[42] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019, pp. 15–21.

[43] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *USENIX Security Symposium (USENIX)*, 2018.

[44] H. J. Tay, K. Zeng, J. M. Vadayath, A. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, A. Doupe, T. Bao, Y. Shoshitaishvili, and R. Wang, "Greenhouse: Single-service rehosting of Linux-based firmware binaries in user-space emulation," in *USENIX Security Symposium 2023*, 2023.

[45] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang *et al.*, "Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 413–430.

[46] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' process and mental models," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–6.

[47] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[48] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares." in *The Network and Distributed System Security (NDSS) Symposium*, vol. 14, 2014, pp. 1–16.

[49] J. Zhang, R. Zhao, and J. Pang, "Parameter and return-value analysis of binary executables," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1. IEEE, 2007, pp. 501–508.

[50] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation." in *USENIX Security Symposium*, 2019, pp. 1099–1114.

[51] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 417–428.

Vendor	Device	Mango				
		TruPoC	TP	Time	TruPoC Bins	Total Bins
D-Link	DIR600	1	0	0:47h	1	48
D-Link	DIR300	1	0	0:44h	1	40
Belkin	F9J1102	52	25	1:46h	4	71
Linksys	WRT320N	22	3	0:07h	4	49
Trendnet	TEW733	4	0	0:50h	2	58
Tenda	N60	109	50	3:25h	8	65
Total	-	189	78	7:39h	20	331

Table 7: Mango analysis on six unique firmware outside the dataset of Karonte and SaTC.

A Additional Experiments

A.1 Additional Firmware Tests

To show our extensibility beyond the Karonte Dataset, we randomly selected 6 firmware samples from different vendors (Shown in Table 7). There was no additional effort or analyses added to Mango to generate these results; they were produced in the same manner as the results discussed in table 1 and table 3. Mango found a significant number of TruPoC's in these firmware samples, including TruPoC's in Belkin and Linksys firmwares which were not tested at all in the Karonte Dataset. Each generated TruPoC was manually analyzed to determine the overall TP rate, which was 41.3%