Finding Taint-Style Vulnerabilities in Linux-based Embedded Firmware with SSE-based Alias Analysis

Kai Cheng¹,², Tao Liu³, Le Guan⁴, Peng Liu³, Hong Li¹, Hongsong Zhu¹, Limin Sun¹

¹Institute of Information Engineering, Chinese Academy of Sciences, China
²School of Cyber Security, University of Chinese Academy of Sciences, China
³The Pennsylvania State University
⁴University of Georgia

chengkai@iie.ac.cn, tul459@psu.edu, leguan@uga.edu, pxl20@psu.edu, {lihong, zhuhongsong, sunlimin}@iie.ac.cn

Abstract

Although the importance of using static analysis to detect taint-style vulnerabilities in Linux-based embedded firmware is widely recognized, existing approaches are plagued by three major limitations. (a) Approaches based on symbolic execution may miss alias information and therefore suffer from a high false-negative rate. (b) Approaches based on VSA (value set analysis) often provide an over-approximate pointer range. As a result, many false positives could be produced. (c) Existing work for detecting taint-style vulnerability does not consider indirect call resolution, whereas indirect calls are frequently used in Internet-facing embedded devices. As a result, many false negatives could be produced.

In this work, we propose a precise demand-driven flow-, context- and field-sensitive alias analysis approach. Based on this new approach, we also design a novel indirect call resolution scheme. Combined with sanitization rule checking, our solution discovers taint-style vulnerabilities by static taint analysis. We implemented our idea with a prototype called EmTaint and evaluated it against 35 real-world embedded firmware samples from six popular vendors. EmTaint discovered at least 192 bugs, including 41 n-day bugs and 151 0-day bugs. At least 115 CVE/PSV numbers have been allocated from a subset of the reported vulnerabilities at the time of writing. Compared to state-of-the-art tools such as KARONTE and SaTC, EmTaint found significantly more bugs on the same dataset in less time.

1 Introduction

With the emerging of the Internet of Things (IoT) technologies, Linux-based embedded firmware is nowadays playing an increasingly important role. For example, almost all the mainstream wireless routers are running Linux-based embedded firmware under the hood. Since wireless routers are often the perimeter defense that separates the information processing and storing requirements (e.g., laptops, smartphones, database servers) in an apartment, a house or a small business office from the Internet, the embedded firmware holds a privileged position in home and small business networks and therefore, they must operate securely. The addition of various IoT devices (e.g., thermostat, smart light, smart lock, smart alarm system) to home and small business networks makes the position held by Linux-based embedded firmware even more vital. Unfortunately, Linux-based embedded firmware still suffers from a number of vulnerabilities in the real world.

To test the real-world Linux-based embedded firmware, static analysis [13,20,35,38] and dynamic analysis [28,43,45] are two basic approaches. Recently, substantial progress has been made on automated dynamic analysis of embedded firmware. For example, Firmadyne [9] has successfully emulated the execution of over 1,900 firmware images. However, existing dynamic analysis techniques are still quite limited. For example, due to reasons such as complex environment dependencies of the firmware, Firmadyne [9] was only able to emulate 23% of the collected 8,591 firmware images.

Being complementary to dynamic analysis, static analysis of embedded firmware has also been attracting an increasing interest in the research community. A unique merit of static analysis is that in many cases it can simply ignore the complex environment dependencies of the firmware while still being able to find many vulnerabilities. In this work, we propose a new static binary analysis approach to find taint-style vulnerabilities in COTS (commercial off-the-shelf) Linux-based embedded firmware. (Because the source code is not available, note that we cannot use any source-code-analyzing tools.) The taint-style vulnerability is a class of vulnerabilities that share the same underlying theme rooted in information flow analysis [42]. More specifically, attacker-controlled data are passed from an input source to a security-sensitive sink without proper check or sanitization. This vulnerability class captures the root cause for many software defects and is often manifested into many common vulnerability types including buffer overflow and format string vulnerability.

Given an embedded firmware image, the standard procedure for detecting taint-style vulnerability consists of four steps. (1) Recover the inter-procedural control flow graph (ICFG) of the program. (2) Identify attacker-controlled
problems with existing work. As we will discuss in detail shortly in Section 2, we found that the existing static binary analysis works on taint-style vulnerability hunting are still very limited in achieving high accuracy, mostly due to the challenges they face in the aforementioned Step 1 and Step 3. Their main limitations can be summarized as follows:

(1) With two alias pointers being dereferenced to different addresses, works (e.g., KARONTE [35]) based on symbolic execution may suffer from high false negative rate.

(2) Works (e.g., Loongchecker [14]) based on VSA (value set analysis) often provide an over-approximate pointer range. As a result, many false positives could be produced.

(3) Existing works for detecting taint-style vulnerability do not consider indirect call resolution, partially due to the associated difficulties. However, based on our study, indirect calls are frequently used in Internet-facing embedded devices (see Section 5.2). By missing many taint propagation links, many false negatives could be produced.

Our solution. We argue that inadequate alias analysis and lack of indirect call resolution are two technical issues for cost-effective embedded firmware analysis. Crossing these barriers will unleash the capabilities of static analysis to find taint-style vulnerabilities. Note that the results of alias analysis can be directly used to recover indirect calls. Therefore, in this work, we propose a precise demand-driven flow-, context- and field-sensitive alias analysis mechanism. We further leverage it to find the alias relationship between the indirect call targets and address-taken functions, yielding an accurate set of indirect-call targets. This is fundamentally different from existing type matching based approaches which suffer from high false-positives. Combined with sanitization rule checking, our solution finds more bugs in less time with fewer false positives.

The proposed SSE-based alias analysis overcomes the drawbacks of existing VSA-based or symbolic-execution-based approaches in two aspects. First, we extend the symbolic values used in symbolic execution to incorporate abstract memory operations, eliminating the need for the information-losing concretization process. Second, our analysis is demand-driven in a sense that only interesting variables need to be traced, avoiding the holistic analysis used in VSA (while localized VSA is possible, it may miss many useful information and lead to bogus links in general). Concretely, given a variable at a particular program point, our approach finds all its aliases along the path by following the use-define and define-use information forwards and backwards. To execute this design, we borrow ideas from access path [12] which represents memory locations by how they are accessed from an initial variable. Since access path is designed for analyzing source code, we specifically accommodate it for binary-only analyses by incorporating our new contributions. Specifically, we propose a new notation called \textit{structured symbolic expression} (SSE) to facilitate the idea. SSE encodes the provenance of a variable using symbolic expression. However, different from symbolic values used in symbolic execution, it not only encodes arithmetic operations, but also considers memory operations such as load and store. As such, multi-level pointer dereferences (e.g., \texttt{x.y.z}) can be expressed in an SSE as a concatenation of symbolic load/store/arithmetic statements with base and offset information. Our algorithm starts from an interesting SSE-encoded variable. When we find a new use or define for any field in the SSE, a new alias SSE is generated by replacing that field. The new alias is then used in a new round of alias searching until a fixpoint is reached. SEE tracking can easily traverse across function boundaries, making inter-procedural analysis straightforward. As such, our approach is flow-, context-, and field-sensitive. This unique design brings about the following desirable benefits. First, alias searching can start at any program location without losing alias information before it. Therefore, it finds a comprehensive list of aliases for a given variable, regardless of the specified starting point. In this way, we do not need to rely on expert input to start analysis from a firmware-specific point near the sink [13, 35]. Rather, we can start from commonly used taint sources such as the \texttt{recv} function. Second, our approach does not need to analyze the program as a whole like VSA. It focuses on a particular variable and only computes over relevant instructions. This on-demand feature allows for more efficient and scalable analysis for complex programs.

We have implemented the proposed SSE-based demand-driven alias analysis and indirect call resolution based on EmTaint. They were integrated into a prototype system called EmTaint to find taint-style vulnerabilities in Linux-based embedded firmware. We evaluated EmTaint with 35 real-world firmware samples. The result shows that EmTaint quickly produces a large number of alerts. Regarding indirect call resolution, EmTaint recovered 12,022 of 12,446 (96.6\%) indirect calls in an hour. Regarding vulnerability discovery, EmTaint discovered at least 192 bugs, including 41 n-day bugs and 151 0-day bugs. Each sample takes an average of 3 minutes. We have reported 151 0-day bugs to the relevant manufacturers for responsible disclosure. 115 of which are confirmed by CVE/PSV at the time of writing. We also conducted a comparison with KARONTE [35] and SaTC [11]. The results of the comparison shown that EmTaint can find more bugs in less time.

Contributions. In summary, we make the following contri-
functions that are indirectly invoked. Given an embedded program, the standard procedure for detecting taint-style vulnerability consists of four steps. We list each step and discuss the key requirements for them as follows.

1) Recover the inter-procedural control flow graph (ICFG) of the program. ICFG is the union of all functions’ CFG by connecting calling edges. Compared with direct calls, recovering indirect calls is considered challenging in static analysis since the calling target is only determined at runtime. Recovering the ICFG is necessary for inter-procedural alias analysis.

2) Identify attacker-controlled sources and security-sensitive sinks. Both the source and sink are typically recognized based on semantics of the relevant functions. For example, the function recv receives data from the network and its second buffer pointer should be tainted. The function system invokes a shell to execute arbitrary commands. Therefore, its parameter is considered as a sink. This step requires the accurate identification of the corresponding functions.

3) In the recovered ICFG, find a path where the taint is propagated from the source to the sink. This step is the most crucial but challenging. As shown in Figure 1, it should cross three barriers – multi-level pointer alias analysis, inter-procedural alias analysis, and taint propagation analysis.

4) Check the constraints of the tainted data at sinks. If not constrained, an alert about the vulnerability is raised.

An ideal solution should achieve high accuracy in all the steps. When high accuracy is not achieved, the analysis usually suffers from two bad consequences: (1) a good portion of the paths found in Step 3 hold one or more bogus links: such paths do not reveal real vulnerabilities, they produce false positives; (2) due to missing links, some vulnerability-revealing paths are not identified: this produces false negatives.

2.2 Limitations of Existing Techniques

Unfortunately, existing works on taint-style vulnerability hunting are still very limited in achieving high accuracy, mostly due to the challenges they face in the aforementioned Step 1 and Step 3.

2.2.1 Alias Issues

Works based on symbolic execution may suffer from high false negative rate. Symbolic execution has been widely used in taint analysis [17, 20, 34, 35]. Concretely, existing works taint the input and propagate the tainted data via forward symbolic execution. An alert is raised when a feasible path is found to have the taint in any of the sinks without sanitization. Apart from the well-recognized problems in symbolic execution (path explosion and timing-consuming constraint solving), the accuracy of these approaches depend on the concretization strategy. Specifically, when a symbolic address is accessed, the symbolic execution engine has to concretize the symbol to allow for continuous execution. Ideally, two alias symbolic addresses should be concretized into the same concrete address. However, it is a non-trivial task to maintain
this information in symbolic execution. With two alias pointers being dereferenced to different addresses, taints cannot propagate, causing a high false negative rate.

To mitigate path explosion issues, existing works [34, 35] use under-constrained symbolic execution [31], in which symbolic execution starts from arbitrary functions, instead of the main entry point. In this way, the distance from the source to the sink is reduced and thus the chance for indirect calls along the path is decreased. However, this approach needs experts to manually specify firmware specific source functions.

Works based on VSA often provide an over-approximate pointer range. VSA has been widely used in many static analysis applications, ranging from recovering binary properties [6, 7] to identifying binary vulnerabilities [14, 32]. The high-level idea of VSA is to maintain a tight over-approximation of the set of numeric values or addresses that each register or variable might hold at a given program point. By providing information about the register values that appear in an indirect memory operand, VSA can determine the addresses that are potentially accessed. This is particularly useful for alias analysis which aims to determine whether two pointers refer to the same address [37]. However, when VSA computes the pointer ranges by following a set of rules (e.g., add operation), it can only provide an over-approximate range. Alternatively, when VSA has no clue to constrain a pointer, the pointer becomes even untrackable. This problem becomes noticeable when it comes to complex programs. Specifically, bogus dependency renders it expensive and impractical for real-world programs [44].

2.2.2 Indirect Call Issues

Existing works for detecting taint-style vulnerability do not consider indirect call resolution, partially due to the associated difficulties. Most related works focus on resolving indirect jumps (e.g., switch statements in C) or simple indirect calls [16, 18, 27, 36], in which no cross-function references occurs. None of them can effectively handle complex situations such as indirect calls implemented by callbacks or function tables. For example, X-Force [30] uses concrete execution to force binary execution without requiring valid inputs or proper environment. However, it is expensive and introduces many infeasible paths. TypeArmor [40] infers the function type information from binary and uses argument number to match callers and callees. However, it suffers from a high false-positive rate because many functions may share the same type. To refine the indirect-call targets, TypeDive [26] leverages multi-layer type information. However, this approach requires the source code.

3 Overview

3.1 Key Insight

Both symbolic execution based and VSA based approaches face challenges to find accurate alias information for real-world programs. The root cause is that both techniques use a store-based approach to represent runtime memory locations, which cannot accurately track indirect memory accesses (i.e., x, y, z) whose address cannot be unambiguously represented. For example, when reading from an unknown location, VSA conservatively models the response as any value (i.e., ⊤), while symbolic execution models the response as an unconstrained symbolic value. Both introduce inaccuracy when this value is later used as an address (e.g., symbolic execution must concretize it using certain strategies). To address this problem, the store-less approach can be used, in which memory locations (or pointers) are represented by how they are accessed from an initial variable (i.e., provenance), instead of a value (e.g., a-Locs in VSA and symbolic or concrete values in symbolic execution). This idea was initially proposed in access path to find aliases in source code [12]. We are inspired by it and incorporate into it our new contributions to handle binaries. The key contribution of this work is the newly proposed structured symbolic expression (SSE), which works at instruction (or IR) level and can represent arbitrary computations, including how a pointer accesses a memory location. Therefore, it does not rely on the source code information and directly handles multi-level pointers. Using SSE, we can derive new aliases of a variable of interests by iteratively traversing the define-use and use-define chain in a binary. At each define or use point, a new alias is derived, represented by another SSE that encodes the corresponding provenance information.

3.2 Architecture Overview

EmTaint takes an embedded firmware image as input and reports potential taint-style vulnerabilities through static analysis. As shown in Figure 2, the proposed system is comprised of four major components.

Firmware preprocessing. The firmware preprocessing module utilizes Binwalk [25] to decompress and extract binaries from firmware and leverages the state-of-the-art reverse engineering tools to extract code and data from the binaries and converts them into intermediate representation (IR). It also builds the control flow graph (CFG) and a partial call graph (CG) that facilitate static analysis (Section 4.1).

SSE-based demand-driven alias analysis. The SSE-based
alias analysis engine is a novel flow-, context- and field-sensitive alias analysis tool (Section 4.2). It is demand-driven, therefore it scales well for complex programs. This component lays the foundation for efficient taint tracking and indirect call resolution mechanisms.

Indirect call resolution. Leveraging the data dependence information provided by the proposed SSE-based alias analysis engine, we further design an indirect call resolution algorithm that checks the data dependence between the referenced function pointers (or function table pointers) and target pointers in indirect call sites (see Section 4.3). Since indirect call resolution allows tainted variables to traverse more function boundaries, it positively impacts the comprehensiveness of the data flow analysis in EmTaint and more importantly substantially improves the discovery of taint-style vulnerabilities. Given the pervasiveness of indirect calls in firmware, the indirect call resolution module is one of the key enablers for the effectiveness of the proposed solution.

Taint-style vulnerability check. The taint-style vulnerability check module is responsible for identifying the potential taint-style vulnerabilities. Specifically, it firstly taints the variables at taint sources that are controlled by attackers (e.g., the recv function). Then, it captures taint propagation via the demand-driven alias analysis module and taint propagation functions (e.g., strcpy). Finally, at security-sensitive sinks (e.g., the system function), if the corresponding variable is tainted and unconstrained, we mark the source-to-sink path as potentially exploitable (see Section 4.4).

4 System Design and Implementation

In this section, we illustrate the detailed design and implementation for each component. The proposed analysis is based on the VEX intermediate representation (IR) [29]. The VEX IR is a popular IR widely used in many program analysis tools, including Valgrind [29] and angr [39]. It is architecture-agnostic, so it can be translated from a number of target machine languages, including x86, ARM, MIPS, and PowerPC. Benefiting from this design choice, our tool is applicable to mainstream architectures used in embedded firmware, including ARM and MIPS.

4.1 Firmware Preprocessing

An embedded firmware image is typically a Binary Large OBject (BLOB), which is a collection of binary data that includes both the (compressed) kernel image and the file system. First, we use Binwalk [25] to recognize, unpack, and extract executables of our interest (e.g., httpd). Then, we use IDA Pro [23] to automatically identify code/functions for the selected executables. However, IDA Pro is not proficient enough to locate some functions that are indirectly called. We therefore developed an IDA plugin to augment this capability. Our observation is that many function pointers are hardcoded in the data segment. Therefore, our plugin scans the data segment and collects all the immediate values as candidates. To confirm a function pointer, a candidate must meet two conditions. One is that its value has to fall into a code segment, and the other is that the instruction sequence where the candidate points to must match the signature of a function prologue. After obtaining all the potential functions, EmTaint loads the binary with APIs provided by angr [39], converts them into VEX IR by using pyvex [4] and generates control flow graphs (CFG) to facilitate further analyses. Note that at this stage, the call graph is incomplete due to the missing calling relationships of indirect calls.

4.2 SSE-based Demand-driven Alias Analysis

In this section, we describe the proposed demand-driven alias analysis. We start with the definition of structured symbolic expressions (SSE), which is the basis of our approach. Then, we explain how to use SSE to find aliases of a given variable/pointer within a basic block. We also show some running examples. Finally, we illustrate how to implement intra- and inter-procedural alias analyses.

4.2.1 Definition of SSE

SSE is a new notation that uses abstract memory model to represent aliases of a variable by encoding the nested provenance information at relevant program points. In Table 1, we recursively defines an SSE expression. Note that since our implementation is based VEX IR, our SSE definition is deeply influenced by its design, in particular the basic statements and memory model. An SSE could be any basic variable or the results of a binary/unary arithmetic operation over basic variables. The basic variable can be either a register (denoted as τreg), a primitive immediate value (denoted as τval), or a memory access (denoted as τmem). The memory access can be a value loaded from memory (denoted as load(expr)) or a value stored to memory (denoted as store(expr)). Here, the expr is a pointer.

We have implemented SSE based on Claripy, a popular SMT solver engine used in angr. Claripy provides a unified way to represent concrete and symbolic expressions. It supports all common arithmetic operations (+, -, *, /, etc.) with an extensible interface [1]. We leveraged this interface to add the support of the memory load and store operations.

An intuitive example. Now, we use an intuitive example to explain how we leverage SSE to find aliases to a given pointer. In the code snippet in Figure 3, there are three instructions. Our goal is to find all aliases to R1 in line 2, which is known to be a pointer. First, we initialize the alias set with R1 in line 2’. By looking backward, we find a definition of

| Table 1: Recursive definition of SSE. |
|--------------------------------------|
| expr ::= expr ‖ expr ‖ expr ‖ var   |
| τreg ::= +, -, *, /, ≪, ≫, !, ... |
| τval ::= r_i                       |
| τmem ::= load(expr) ‖ store(expr)  |
Algorithm 1 Alias update within a basic block

1: procedure TraceBlock($PRE_f$, $SUC_b$)  
2: \textit{TARGET}_f, $TARGET_b$ $\leftarrow$ GetCurrentTarget()  
3: $IN_f$ $\leftarrow$ $PRE_f$ $\cup$ $TARGET_f$  
4: $IN_b$ $\leftarrow$ $SUC_b$ $\cup$ $TARGET_b$  
5: do  
6: \textit{NEW}_f, $NEW_b$ $\leftarrow$ ForwardUpdate($IN_f$)  
7: \textit{OUT}_f $\leftarrow$ $OUT_f$ $\cup$ $NEW_f$  
8: $IN_b$ $\leftarrow$ $IN_b$ $\cup$ $NEW_b$  
9: $NEW_f, NEW_b$ $\leftarrow$ BackwardUpdate($IN_b$)  
10: $OUT_b$ $\leftarrow$ $OUT_b$ $\cup$ $NEW_b$  
11: $IN_f$ $\leftarrow$ $NEW_f$  
12: $IN_b$ $\leftarrow$ $\emptyset$  
13: while new aliases can be generated  
14: return $OUT_f$, $OUT_b$

R1 in line 1, which gets its value by loading from memory R3+0x8. Therefore, by replacing R1 with load(R3+0x8), we add load(R3+0x8) in line 1’ to the alias set. Now, if we look forward from line 1, we find a usage of R3 in line 3. That is, the value of R3 is stored in the memory R6+0x4. Therefore, by replacing R3 with store(R6+0x4) in load(R3+0x8), we add load(store(R6+0x4)+0x8) in line 3’ to the alias set. By doing so back and forth iteratively along the define-use and use-define chain, we can eventually reach a fixpoint of the alias set for the given pointer. The full SSE update rules within basic blocks and the algorithm for intra- and inter-procedural analyses are explained in the following sections.

![Figure 3: An intuitive simple example to illustrate SSE-based alias analysis](image)

By using an SSE to encode pointer provenance information, complex data structures (e.g., x,y,z) can be naturally traced. Due to the on-demand feature, it does not need to analyze the program as a whole. Instead, we can focus on a particular variable and only compute over relevant instructions. Therefore, our approach scales well for complex programs.

4.2.2 Alias Update Algorithm within a Basic Block

In this section, we describe how to find aliases for a given pointer within a basic block. Our algorithm, called TraceBlock, is shown in Algorithm 1. As mentioned before, for a given pointer, we search both forwards and backwards to enrich the alias set. As such, for each basic block, we maintain two different sets of aliases for forward and backward results respectively. When running TraceBlock over a block, it takes two parameters – one forward alias set from all $currentblock$’s predecessors (denoted as $PRE_f$) and one backward alias set from all $currentblock$’s successors (denoted as $SUC_b$). At the initial basic block, $PRE_f$ and $SUC_b$ are empty. In line 2, GetCurrentTarget obtains the manually specified target pointers of the current basic block (e.g., the taint source). Then, in line 3, $IN_f$ is initialized to be the union of $PRE_f$ and $TARGET_f$. In line 4, $IN_b$ is initialized to be the union of $SUC_b$ and $TARGET_b$. Here, $IN_c$ contains all the SSEs needing forward tracking and $IN_b$ contains all the SSEs needing backward tracking. Forward and backward tracking are implemented via ForwardUpdate and BackwardUpdate respectively. ForwardUpdate takes $IN_f$ as input and generates new forward SSEs, which are merged into $OUT_f$. ForwardUpdate may also return new backward SSEs, which are merged into $IN_b$ for further processing. BackwardUpdate takes $IN_b$ as input and generates new backward SSEs, which are merged into $OUT_b$. BackwardUpdate may also return new forward SSEs, which are merged into $IN_f$ for further processing. The algorithm runs iteratively and terminates when no new alias SSE can be generated (line 5-12). The output of the algorithm, $OUT_f$ and $OUT_b$, are used as parameters for tracking succeeding basic blocks and preceding basic blocks respectively (see Section 4.2.3 for more details). In the following, we detail the forward update and backward update processes respectively.

**Forward update.** During forward analysis, $EmTaint$ traverses instructions following the instruction flow. In the simplest form, for a live variable $v$ in the current SSE, if $v$ is on the RHS (right hand side) of an assignment statement $d = v$ (i.e., $v$ is used in a define-use chain), a new SSE is generated by replacing the variable $v$ in the original SSE with the variable $d$. Therefore, the new SSE becomes an alias to the old one, and the new SSE needs to continue to be tracked forwards. The full SSE update rules following the define-use chain are listed in entries 1-7 in Table 2. In the table, each entry is represented as $statement \xrightarrow{rn} expr.replace(rm, rj)$, where $statement$ is the encountered instruction and $expr$ is the tracking SSE. In the $statement$, if $rm$ is true and $rn$ exists in $expr$, then $rn$ should be replaced with $rj$. Note that this process should be conducted over all the live variables in the current SSE.

As mentioned before, a forward update operation following the define-use chain can also generate SSEs needing backward tracking. Specifically, one type of assignment statement is in the form of $store(d) = v$ and $v$ is a pointer. For such type, in addition to continuing forward tracking, the new SSE needs to be tracked backwards also to find the definitions of the store address, i.e., $d$, which is because once such a definition is found, a new alias can be generated.

**Backward update.** During backward analysis, $EmTaint$ traverses instructions following the instruction flow reversely. In the simplest form, for a live variable $v$ in the current SSE, if $v$ is on the LHS (left hand side) of an assignment statement $v = u$ (i.e., $v$ is defined in a use DEFINE chain), a new SSE is generated by replacing the variable $v$ in the original SSE with...
the variable \( u \). Therefore, the new SSE becomes an alias to the old one. Now that \( u \) is live in both directions, the new SSE needs to be tracked both backwards and forwards to find the definitions and uses of the variable \( u \). The full SSE update rules following the use-define chain are listed in entries 8-13 in Table 2.

Different from forward tracking, in backward tracking, we should not only find definitions of a variable \( v \) (i.e., \( v = u \) as explained before), but also find its uses (i.e., \( d = v \)). If a use is found, we follow entries 1-6 in Table 2 except for entry 7 to update new SSEs. However, the new SSE can only be tracked forward, not backward, because the variable \( d \) used to update the original SSE is live only in forward direction. One type of assignment statement is in the form of \( \text{store}(d) = v \) and \( v \) is a pointer. Note that for such type, in addition to forward tracking, the new SSE needs to be tracked backwards also, to find the definition of store address (i.e., \( d \)).

**Conditions to kill an SSE.** Each variable has its own live scope. The liveness of the corresponding register (i.e., \( \tau_{reg} \)) is killed if we encounter a register assignment statement. If this happens in forward/backward analysis, the whole SSE with this register is killed. This corresponds to entry 14 in Table 2. Similarly, the liveness of the corresponding memory access (i.e., \( \tau_{mem} \)) is killed if we encounter a store statement. If this happens in forward analysis, the whole SSE with this variable is killed. This corresponds to entry 15 in Table 2.

**A non-trivial running example.** Following Algorithm 1, we are able to find alias information for more complex programs. We show such an example in Figure 4, in which \( R1 \) in line 3 and \( R0 \) in line 5 are actually aliases to each other.

Assuming that the analysis begins at line 3, in which \( R1 \) is load from the address \( R3+0x8 \). To find aliases of \( R1 \), \( EmTaint \) first initializes an SSE as \( \text{load}(R3+0x8) \) in line 3'. In the first round of forward analysis, it does not find any use of

### Table 2: Update rules for structured symbolic expressions

| SSE update with define-use chain | SSE update following use-define chain |
|---------------------------------|-----------------------------------|
| \( 1) \) \( ri \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, ri\right) \) | \( 8) \) \( ri \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, ri\right) \) |
| \( 2) \) \( \text{Binop}\left(rj, rm\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, \text{Binop}(ri, rm)\right) \) | \( 9) \) \( \text{Binop}\left(rj, rm\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, \text{Binop}(ri, rm)\right) \) |
| \( 3) \) \( \text{ITE}\left(rj, rm, rm\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, \text{ITE}(ri, rm, rm)\right) \) | \( 10) \) \( \text{ITE}\left(rj, rm, rm\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, \text{ITE}(ri, rm, rm)\right) \) |
| \( 4) \) \( rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, ri\right) \) | \( 11) \) \( rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj, rm\right) \) |
| \( 5) \) \( \text{Load}\left(rj\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{load}(rj)\right) \) | \( 12) \) \( \text{Load}\left(rj\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{load}(rj)\right) \) |
| \( 6) \) \( \text{Store}\left(ri\right) = rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{load}(rj)\right) \) | \( 13) \) \( \text{Store}\left(ri\right) = rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{load}(rj)\right) \) |
| \( 7) \) \( \text{Load}\left(rj\right) \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{store}(rj)\right) \) | \( 14) \) \( rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(rj\right) \) |
| \( 15) \) \( \text{Store}\left(ri\right) = rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{store}(rj)\right) \) | \( 16) \) \( \text{Store}\left(ri\right) = rj \xrightarrow{\text{d}} \text{expr}\text{.replace}\left(\text{store}(rj)\right) \) |

| \( \downarrow \) \text{: The statement } \text{ri} = \text{ITE(ri, rm, rm)} \text{ denotes that if } \text{rj} \text{ is true, } \text{ri} = \text{rm} \text{, otherwise, } \text{ri} = \text{rm}. \downarrow | \( + \) \text{: Existing load(ri) in expr occurs after the newly encountered store(ri) statement.} |
sal of the CFG (line 2), in which a block is visited after all its successor blocks have been visited. Then, the procedure FINDALIAS is used to actually analyze each block. We use the same procedure over the WorkList forwards and backwards, until no new alias can be found.

Inside FINDALIAS, TraceBlock is applied to analyze aliases within each basic block BB (line 16). Then the results are merged to the successors and predecessors of the current BB (line 17-18). However, if the current basic block is a callsite, we need to incorporate inter-procedural analysis. Note that in our design, a call instruction is treated as a separated basic block, which may differ from the definition of basic blocks in other tools. Specifically, FINDALIAS collects modifications (MOD) and references (REF) to memory locations that are accessed directly or indirectly through parameters, return values, or global pointers in the callee (line 11-14). In other words, we summarize the modifications and references to memory locations by callee. To do so, at first, the algorithm checks the SSEs in the forward alias set PRE_f and backward alias set SUC_b of the callsite BB, extracts the root pointer of each SSE (e.g., the root pointer of load(80+0x8) is R0), and stores them in a set called PTRs (line 12). Then, we use a transfer function (TRANSFERFUNC) [12] to actually get MOD and REF. Our implementation of transfer function is quite standard so we omit the details. Finally, with MOD and REF, FINDALIAS updates SSEs in PRE_f and SUC_b of the callsite BB with the definitions in MOD and REF (line 14).

Back to ANALYZEFUNCTION, after using FINDALIAS iteratively to reach a fixpoint, the results of the current function are merged to the caller. Specifically, SSEs associated with arguments or global pointers from the entry block’s OUT_b are merged into SUC_b of the caller’s call site (line 7), and SSEs associated with return value from the exit block’s OUT_f are merged into PRE_f of caller’s return site (line 8).

4.3 Indirect Call Resolution

EmTaint resolves indirect calls based on dependencies between the aliases of indirect call targets and the aliases of function pointer references (or function table pointers). First, EmTaint identifies all the indirect call instructions by traversing the disassembled code. EmTaint treats all branch instructions with a register as operand to be indirect calls (e.g., blx r0 in ARM and jalr $t9 in MIPS). EmTaint then uses IDA Pro to filter call instructions whose targets can be recognized. The remaining ones were treated as unresolved indirect calls. Since almost all the call instructions in MIPS use a register as operand, MIPS programs have a considerably larger number of indirect calls than ARM programs (see Table 5).

Second, EmTaint finds all address-taken functions by scanning both the code segment and data segment, and locates all the references to the address-taken functions. There are mainly two ways to reference a function. A function pointer which contains the address of the function, can be used directly as an operand in an indirect call. We denote this kind of references as f ptr and show an example in line 2 of Listing 1. A function table pointer which contains the address to a function table, can be used indirectly (i.e., by dereferencing the function table first) to get the real function address. We denote this kind of references as d ptr and show an example in line 7 of Listing 1. In this example, an entry in the table has two fields—a string pointer to the function name and the actual function pointer.

Algorithm 2  Intra- and inter-procedural alias analysis

```plaintext
1: procedure ANALYZEFUNCTION(CFG)  
2:     WorkList ← Postorder(CFG)  
3:     do  
4:         FINDALIAS(WorkList.reverse())  \textit{forward}  
5:         FINDALIAS(WorkList)  \textit{backward}  
6:     while new aliases can be generated  
7:         MERGE(Callers, ExitBB.OUT_b)  
8:         MERGE( callers, ExitBB.OUT_f)  
9:     procedure FINDALIAS(WorkList)  
10:     for BB ∈ WorkList do  
11:         if BB is a callsite then  
12:             PTRs ← GETROOTPTR(PRE_f, SUC_b)  
13:             MOD, REF ← TRANSFERFUNC(PTRs, callee)  
14:             UPDATECONTEXT(PRE_f, SUC_b, MOD, REF)  
15:         else  
16:             OUT_f, OUT_b ← TraceBlock(PRE_f, SUC_b)  
17:             MERGE(BB, successors, BB.OUT_f)  
18:             MERGE(BB, predecessors, BB.OUT_b)  
```

Listing 1: Indirect call resolution example.

Third, EmTaint performs alias identification. Our goal is to find dependence between the indirect call targets (e.g., fp1 and fp2 in line 5 and 13) and the original references (e.g., fptr and dptr in line 2 and 7). EmTaint back tracks the indirect call targets to find its all aliases. At the same time, EmTaint finds the aliases of fptr and dptr through both forward and backward analyses. Then, EmTaint collects all the alias SSEs in the entry block and exit block of every function, checks the data dependency between aliases of fptr/dptr and aliases of indirect call target, and updates them. We denote the alias SSEs of the indirect call targets (e.g., fp1 and fp2 in the listing) as CExpr (call target expression), and the alias SSEs of the fptr or dptr as PExpr (pointer expression).
Finally, \textit{EmTaint} matches the aliases to eventually resolve indirect calls. In the simplest form, a \textit{CTexpr} is an alias to \textit{Pexpr}. \textit{EmTaint} can calculate the call target directly from \textit{CTexpr}. This often indicates an indirect call by \textit{fptr}. If the \textit{CTexpr} can be represented by \textit{load(dptr + i*stride)}, where the \textit{dptr} is the address of a function table, the \textit{stride} is a constant, and \textit{i} is an index, it strongly indicates a function table deference. \textit{EmTaint} reads values from addresses \textit{dptr + i*stride}, where the index \textit{i} starts at 0 and increases by 1 at a time. It terminates when the retrieved value is greater than the maximum address of the function table. If the returned value is a legitimate function address, \textit{EmTaint} adds it to the set of indirect call targets.

For the indirect call targets that cannot be accurately traced to a specific \textit{fptr} or \textit{dptr}, we indirectly find their dependencies. Our observation is that \textit{fptr} and \textit{dptr} are often stored in a multi-level data structure whose root pointer is a global pointer (denoted as \textit{gptr}). Therefore, we often find \textit{store(gptr)} or \textit{store(gptr)} in \textit{Pexpr}. To indirectly call the target, the program also needs to refer to the same global pointer before the callsites. If the \textit{CTexpr} can be simplified to \textit{load(gptr)} where the same \textit{gptr} is used in the \textit{Pexpr}, the indirect call target is immediately recovered by using \textit{fptr}. If the \textit{CTexpr} can be simplified to \textit{load(load(gptr) + i*stride)} where the same \textit{gptr} is used in \textit{Pexpr}, by replacing \textit{load(gptr)} in \textit{CTexpr} with \textit{dptr} (since \textit{dptr} has an alias SSE \textit{load(gptr)}), a new SSE \textit{load(dptr + i*stride)} can be generated. Then, \textit{EmTaint} resolves the indirect call targets using \textit{load(dptr + i*stride)}.

\subsection{Taint-style Vulnerability Check}

The taint-style vulnerability is caused by the lack of security sanitization on security-sensitive sinks whose data can be propagated from attacker-controlled sources. Our goal is to track the tainted data from sources through the proposed alias analysis, examine the constraints of the tainted data, and determine whether the tainted data can be propagated to sinks. Note that our analysis is performed on the improved CFG where indirect calls have been recovered.

\textbf{Taint Source and Sink.} In embedded firmware analysis, \textit{EmTaint} treats the data received from network as taint sources. In C code, sources include \textit{recv/recvfrom} and other library functions, such as \textit{read/fgets}, which may read data from a local file or network. For these functions, \textit{EmTaint} performs a backward data flow analysis on its file descriptor parameter. If the parameter data depends on a specific file, we remove it from the source list. Particularly, \textit{EmTaint} identifies function \textit{getenv} as a source. The sinks include string copy library functions (e.g., \textit{strcpy}, \textit{memcpy}) and command execution library functions (e.g., \textit{system}, \textit{popen}). Table 8 in Appendix A shows all sources and sinks used in \textit{EmTaint}.

\textbf{Library Function Summaries.} In taint analysis, the tainted data may be propagated to library functions. \textit{EmTaint} adopts “function summaries”, which describes what variables are tainted and how these tainted variables are propagated within a function, to handle common string functions. We have implemented summaries for 29 common functions from the Standard C Library (Table 9 in Appendix), such as the function \textit{strcpy(*dest,*src)}, in which the taint is propagated from argument \textit{src} to argument \textit{dest}.

\textbf{Taint propagation.} The taint analysis is implemented based on the SSE-based demand-driven alias analysis. Therefore, taints are propagated along with the alias analysis. For a given source, \textit{EmTaint} taints relevant pointers depending on the semantic of the source function, which can either be function parameters or a return value. Then, \textit{EmTaint} tracks the tainted SSE to find its aliases and propagates the taint to another following arithmetic operations (i.e., \textit{a=b+1}), library functions listed in Table 9 in Appendix B, and loop copy functions. This is quite standard in all related works. During taint analysis, \textit{EmTaint} also collects constraints for the tainted SSE (e.g., \textit{x<4} where \textit{x} is tainted). Whenever the tainted SSE reaches a sink, both the sink’s information (e.g., sink’s function name, buffer size) and the corresponding tainted SSE with its constraints are collected and passed to the taint-style vulnerability check module for further analyses.

In forward analysis, when the tracked SSE contains the parameters of the callee, or when the root pointer of the SSE is a global pointer, \textit{EmTaint} goes into the callee to continue the taint analysis. As an optimization, for each analyzed function, \textit{EmTaint} creates a function summary that describes tainted input data sets and the corresponding tainted output data sets. When \textit{EmTaint} encounters the same function, if the tracked SSE is in the tainted input data sets, the taint is propagated directly based on the summary. Otherwise, \textit{EmTaint} analyzes the function again and complements the existing summary.

However, not all aliases should be tainted following the alias analysis. For example, the buffer pointer tainted in \textit{recv} should not propagate backwards when \textit{recv} has not invoked yet. \textit{EmTaint} determines whether a new alias SSE should to be tainted based on what we call taint-trigger points. The taint-trigger points are a class of callsites, which satisfies the property that the callee’s argument pointer is tainted after the callee is executed. \textit{EmTaint} tracks the taint-trigger point for each alias. Whenever an alias is propagated over its taint-trigger point in forward analysis, the alias is tainted. In contrast, whenever an alias is propagated across its taint-trigger point in backward analysis, the alias is untainted. At sinks, we only consider tainted SSEs to detect taint-style vulnerabilities.

\textbf{Constraint checking.} Constraint checking is the last step to find potential bugs. We follow a quite standard approach. Specifically, at sinks, \textit{EmTaint} checks the associated tainted SSE parameters. If it contains a symbolic constraint (i.e., \textit{str_len < s}), \textit{EmTaint} does not report an alert. Here, the \textit{str_len} is the length of tainted string. Otherwise, \textit{EmTaint} solves the constraints to get its minimum value \textit{min_len}. 

9
For memcpy-like functions, if they copy data to stack buffers, EmTaint retrieves the address of the destination buffer. If \( \text{min\_len} \) is larger than the distance from the buffer address to the top of the stack or there is no constraint on the tainted SSE, an alert is raised. Likewise, for the command execution library functions (e.g. system), EmTaint reports an alert if there is no constraint on the tainted SSE.

5 Evaluation

We have implemented a prototype of EmTaint, which consists of about 24,000 LoC in Python. We evaluated EmTaint from three aspects. (1) How effective is it in uncovering real-world taint-style vulnerabilities in embedded firmware (Section 5.1)? (2) How effective is it in indirect call resolution? To what extent does the indirect call resolution play a role in improving vulnerability discovery (Section 5.2)? (3) How effective is it compared with the state-of-the-art tools (Section 5.3)?

Experiment setup. Our evaluation was conducted against 35 different firmware samples from six major vendors of network embedded systems: Cisco, D-Link, NETGEAR, TRENDSnet, TP-Link and Tenda. These samples were manually downloaded from the vendor’s official websites. In order to compare with the existing tools head-to-head, some samples are outdated. However, as we will explain later, all the identified bugs were tested against the latest firmware versions before reporting them to the vendors. Table 3 summarizes the information about each sample, including its vendor, firmware version, architecture, the analyzed Internet-facing binary, its size, etc. This sample set covers three architectures, ARM32, MIPS32 and MIPS64, which are the mainstream architectures used in embedded devices. Note that since our approach works based on VEX IR, it can be easily extended to support other architectures. All the experiments were conducted on a Ubuntu 18.04.4 LTS OS running on PC with a 64-bit 8-core Intel(R) Core(TM) i7-8550U CPU and 24 GB RAM.

5.1 Vulnerability Discovery

Our current prototype, EmTaint, checks unsafe data copy functions (e.g., strcpy) and command execution functions (e.g., system). Therefore, EmTaint reports vulnerabilities related to buffer overflow or command injection. We summarize the results in Table 3. In total, EmTaint reached 9,346 different sinks where there are tainted parameters to the sink functions. 1,887 of them were identified as alerts because there is no constraint on the tainted SSE, an alert is raised. Likewise, for the command execution library functions (e.g. system), EmTaint reports an alert if there is no constraint on the tainted SSE.

True-positive evaluation. Given the large amount of raised alerts, it is important to understand how many of them are true positives. However, this would require substantial human efforts to manually craft a proof-of-concept (PoC) for each of them on real devices. Reverse-engineering each sample is also impractical because we have no ground truth of the indirect calls. Therefore, we adopted random sampling in an attempt to obtain an estimated true positive rate. Specifically, we acquired 10 physical devices and randomly sampled 100 alerts out of 971 that correspond to these 10 devices. Then, we manually analyzed these alerts. An alert is confirmed as a true positive if (1) it matches a known vulnerability or (2) can be verified by successfully constructing a PoC on the physical device. Table 4 shows the results. Out of 100 alerts, we confirmed 86 true positives, including 13 known vulnerabilities and 73 successfully constructed PoCs. This indicates the high true positive rate of our approach. For the rest, which are false positives, we attribute them to either inaccurate indirect call resolution or failing to find security checks (see Section 6 for details).

False negative evaluation. To evaluate the false negative rate (i.e., the rate of failing to find a bug) of our prototype, we collected 42 known vulnerabilities with exposed details from exploit-db and MITRE CVE for the 35 firmware samples in our dataset. EmTaint discovered 41 of them (see Table 10 in Appendix). The missing one (CVE-2019-6989) is a buffer overflow bug caused by unsafe copy using loop, which our tool does not check currently. We will extend our tool in the future to support more kinds of security checks at sink.

N-day and 0-day vulnerabilities. In addition to verifying alerts through sampling as mentioned before, we have been working hard to verify more alerts. We prioritize alerts related to physical devices that we have access to. At the time of writing, we have confirmed a total of 41 n-day vulnerabilities and 151 0-day vulnerabilities, 115 of which have been assigned with CVE/PSV numbers. As mentioned before, the 41 n-day vulnerabilities were matched from exploit-db [3] or MITRE CVE [2], which correspond to 120 alerts. In Table 10 of Appendix, we list how the identified 41 n-day vulnerabilities are distributed among the public exposure IDs. Note that one CVE ID may correspond to multiple alerts. For example, CVE-2019-13278 describes a command injection vulnerability, which can be triggered at 33 different sinks.

To confirm 0-day vulnerabilities, we tried to craft PoCs against the latest firmware versions of physical devices. In total, we confirmed 151 0-day vulnerabilities, including 38 command injection bugs and 113 buffer overflow bugs. Table 11 in Appendix summaries these bugs and lists relevant CVEs/PSVs. As mentioned before, some of the 35 samples are old-versioned for comparative evaluation, therefore, we found that lots of alerts generated by EmTaint have been fixed in the latest version. These bugs were probably found by the vendor themselves, therefore no public detail is available. For example, EmTaint produced 119 alerts in NETGEAR R7800 v1.0.2.32, but we only verified one 0-day vulnerability in its latest version v1.0.2.68. By reverse-engineering both samples, we found that many bugs were fixed by replacing strcpy with strncpy and setting the string length parameter to a constant.
5.2 Indirect Call Resolution and Its Importance

Indirect call resolution. Our approach was able to resolve 12,022 indirect call targets (out of 12,446) and the average execution time per sample is about 96 seconds. We also calculated the total number of target functions called at indirect call sites during this process, which is listed in the column I-Call targets. In total, our tool added 14,920 target functions that are called indirectly, which substantially improved taint-style vulnerability discovery. Since we do not have the ground truth for indirect call targets, we cannot accurately evaluate the failed resolutions. However, by reverse-engineering some samples, we did find some missed targets because some address-taken functions were not recognized. Interestingly, our tool also resolved some indirect call targets to be NULL. It turned out that the firmware later correctly performs checking before dereferencing them. Therefore, our resolution was correct.

Importance of indirect call resolution. We analyzed the same 35 samples with and without indirect call resolution. Due to page limit, we show the results of 10 representative samples with real devices to conduct this comparative experiment. The results are shown in Table 6. We illustrate how the indirect call resolution enhances taint analysis and further improves vulnerability discovery from four metrics: the number of covered basic blocks, tainted basic blocks, tainted sinks, and generated alerts. As clearly shown by the results depicted in Table 6, the number of covered basic blocks increased from 356,585 to 412,688, and the number of tainted basic blocks increased from 19,218 to 56,488. This means that indirect call resolution enables EnTaint to propagate the tainted data to functions where they were not able to reach previously and taints more variables which were missed before subject to indirect calls. The number of tainted sinks increased from 1,234 to 3,710, which proves that indirect call resolution makes it possible for tainted data to arrive unsafe sinks that were unreachable before. What are the benefits of applying indirect call resolution from the perspective of vulnerability discovery? As shown in the table, 763 more alerts were produced after resolving functions that were invoked indirectly. These alerts in turn correspond to 131 real bugs we introduced before.
For different binaries, indirect call resolution has varied impacts on vulnerability detection. In most cases, indirect call resolution helps report more alerts. For Cisco, NETGEAR, Tenda, and TP-Link in our dataset, almost all alerts were discovered with the help of indirect call resolution. For example, our prototype produced 0 alert originally without indirect call resolution and 120 alerts in total afterwards for product R8000 and AC9V3.0. There is no change in the amount of alerts in firmware sample TEW827RU, which can be attributed to the fact that the propagation of tainted data from source to sink does not involve any indirect calls in this sample.

To further evaluate the effectiveness of indirect call resolution, we manually analyzed the 162 real bugs (11 n-day and 150 o-day) in the 10 samples. Among them, the trigger path of 131 bugs involves indirect calls. Although we do not have the ground truth of indirect call targets in any of these samples (indirect call resolution is widely believed to be difficult, even if the source code is available) and thus cannot evaluate the accuracy of the recovered targets, these 131 real bugs demonstrate the importance of indirect call resolution.

5.3 Comparison with KARONTE and SaTC

Three works are closely related to ours, including CodeSonar [21], KARONTE [35] and SaTC [11]. However, CodeSonar is a commercial product of GrammaTech and we were unable to use it in our evaluation. KARONTE and SaTC are both open-sourced. They perform taint analysis to detect the taint-style vulnerability in embedded firmware through the under-constrained symbolic execution on top of angr [39].

Due to the very similar scope, we reused the dataset [24] released by KARONTE, which includes 49 firmware samples from 4 embedded vendors (i.e., NETGEAR, D-Link, TP-Link and Tenda). In fact, SaTC also used this dataset. This made our experiment a head-to-head comparison.

Table 7 shows the results, where the information for KARONTE and SaTC are copied directly from the original papers [11,35]. To be fair, we adopt the same definition of true positive in KARONTE (cf. §X.D in [35]). Specially, an alert is a true positive if the tainted data that reaches the sink is provided by the user. This applies to SaTC too. KARONTE took approximately 451 hours to produce 74 alerts, among which 46 were true positives; SaTC took approximately 459 hours to produce 2,084 alerts, among which 683 were true positives; EmTaint reported 1,583 alerts in less than 4 hours, 1,518 of which were true positives. The result shows that EmTaint can find more true positives in less time than KARONTE and SaTC. Because of the improvement over existing work, our tool found 22 new 0-day vulnerabilities in the already extensively tested samples. They are included as part of the 151 0-day vulnerabilities we mentioned before.

Apart from the lack of indirect call resolution and less accurate alias analysis, we found that KARONTE and SaTC frequently raise false alerts because of the incorrectly specified taint sources. Specifically, due to the path explosion problem of symbolic execution, they cannot directly use the commonly recognized taint sources such as recv. Instead, to shorten the path from the sources to sinks, KARONTE infers the taint source through the interaction between the back-end programs in the firmware with a preset list of network-encoding strings (e.g., “soap” or “HTTP”), and SaTC infers the taint sources by the keywords shared between the front-end files and the back-end programs. When these keywords are unrelated to user inputs, false positives occur. In contrast, EmTaint starts taint analysis directly from the commonly used sources summarized in Table 8, which helps us achieve much higher true positive rates. We discuss false positives introduced by EmTaint in Section 6.

6 Limitations

In this section, we discuss the limitations of the proposed approach. SSE, the fundamental technique of the proposed system, works well to track pointer between indirect memory accesses. However, it falls short when the pointer involves bitwise operations or the offset of the indirect memory access is not a constant. As such, false negatives could occur for aliases introduced by these operations. Further study is needed to improve SSE to handle this situation.

Second, the current prototype supports discovering buffer overflow and command injection vulnerabilities. In our experiment, most discovered buffer overflow bugs were stack based. Our tool stayed silent in the presence of many heap overflow bugs (false negative). To more accurately predict...

---

**Table 6:** The impact of indirect call resolution on vulnerability discovery.

| Model            | Ana. Block | w/o I-Call Resolution | Tainted Block | Tainted Sink | Alerts | I-Call Resolution | Tainted Block | Tainted Sink | Alerts |
|------------------|------------|-----------------------|---------------|--------------|--------|--------------------|---------------|--------------|--------|
| Cisco RV320      | 74,778     | 976                   | 98            | 16           |        | 89,915             | 16,307        | 1,450        | 335    |
| Cisco RV130      | 26,210     | 1,234                 | 87            | 2            |        | 30,919             | 4,585         | 172          | 84     |
| D-Link DIR-825   | 19,661     | 1,994                 | 161           | 18           |        | 21,169             | 2,553         | 198          | 36     |
| D-Link DAP-1660  | 34,741     | 2,303                 | 100           | 8            |        | 34,881             | 2,326         | 106          | 12     |
| TRENDnet TEW632BRP | 10,756   | 1,556                 | 126           | 13           |        | 11,672             | 1,953         | 149          | 26     |
| TRENDnet TEW827RU | 21,348   | 2,788                 | 286           | 142          |        | 22,714             | 2,860         | 289          | 142    |
| NETGEAR R7800    | 29,640     | 2,382                 | 118           | 8            |        | 34,741             | 2,303         | 118          | 8      |
| NETGEAR R8000    | 46,053     | 1,997                 | 166           | 0            |        | 56,907             | 7,373         | 428          | 36     |
| TP-Link WR940N4V4 | 59,786   | 1,987                 | 50            | 1            |        | 69,759             | 7,258         | 225          | 31     |
| Tenda AC9V3.0    | 23,612     | 2,901                 | 42            | 0            |        | 42,490             | 6,056         | 172          | 84     |
| **Total (10)**   | 356,585    | 19,218                | 1,234         | 208          |        | 412,688            | 56,488        | 3,710        | 971    |
heap overflows, precise analysis is needed to determine the size of the dynamically allocated heap buffer.

Lastly, our tool generates false positives (14% based on our evaluation). Although false positives is fundamental to all static approaches, we explain the root causes specific to our method and discuss how to improve on it. First, our tool misses some constraint checks already exists in the firmware. This is because our prototype analyzes library functions by manually generating function summaries. This leads to missing security checks that do occur in library functions without summaries. Generating more summaries can mitigate this limitation. Second, our tool may incorrectly recover indirect calls that do not exist. This leads to infeasible paths. Using dynamic analysis to filter the results (e.g., dynamic symbolic execution can be used to solve the path constraints) might help reduce this kind of false positives. We leave this as our future work. Third, $EmTaint$ uses the `read` function as a taint source. However, data read from local files actually cannot be manipulated by attackers. Although $EmTaint$ filters obvious read operations from local files, there are cases that $EmTaint$ cannot distinguish statically.

7 Related Work

Vulnerability discovery using static analysis. In static binary analysis, most work detects vulnerabilities through data flow analysis. Balakrishnan et al. first proposed using VSA to achieve flow-, context-sensitive data flow analysis [8] and further commercialized the idea to implement a product called CodeSurfer/x86 [5]. Similar work has been proposed to find taint-style vulnerabilities using VSA [14, 32, 33]. However, VSA cannot achieve high field sensitivity in the multi-level pointers when the address of memory access cannot be concretized. Some work [11, 17, 34, 35] was proposed to discover taint-style vulnerabilities through tracking data-flow information with symbolic execution. Among them, $KARONTE$ [35] is a static analysis framework for embedded firmware that can discover vulnerabilities due to multi-binary interactions. The authors achieve this goal by modeling and tracking multi-binary interactions. $SaTC$ [11] also performs taint analysis to discover bugs. It utilizes shared keywords related to user input in the front-end and back-end to infer the taint source.

Ivan Gotovchits et al., proposed an approach for collecting path- and context-sensitive data-flow information called $\mu$flux. $\mu$flux can find taint-style vulnerabilities by static property checking [20]. In order to find more taint-style vulnerabilities, D$Taint$ [13] adopts pointer alias analysis to improve the data flow analysis and utilizes data structure similarity matching to construct data dependence between functions invoked by indirect calls. However, D$Taint$ lacks accuracy and efficiency in data flow analysis. Most of the static approaches for binary cannot maintain field-sensitivity for data-flow analysis. All the aforementioned works that detect taint-style vulnerabilities did not perform indirect call resolution, which we have demonstrated to be critical. $EmTaint$ addresses both problems.

Alias analysis. Alias analysis is a long-term research topic in static binary analysis. Debray et al. [19] proposed an inter-procedural flow-sensitive pointer alias analysis for x86 executables, which is context-insensitive. Guo et al. [22] presented the first context-sensitive points-to analysis for x86 assembly code. However, this approach is only partially flow-sensitive. Reps et al. [37] utilized value-set analysis (VSA) to identify pointer alias through tracking memory accesses in x86 executables. Chong et al. [15] presented a flow-sensitive algorithm for instruction-level alias analysis on ARM executables, which is context-insensitive. We proposed a new alias analysis technique based on structured symbolic expressions (SSE). Our implementation is based on VEX IR. Therefore it is architecture-neutral and we have applied our tools against both ARM and MIPS executables. To the best of our knowledge, this is the first work that simultaneously achieves demand-driven, flow-, context- and field-sensitive alias analysis for binary. SSE shares the spirit of access path [12]. However, access path targets source code, not the binary.

Vulnerability discovery using dynamic analysis. Fuzzing has been widely adopted as a dynamic analysis technique to uncover vulnerabilities in embedded devices. RPFuzzer [41] is a fuzzing framework specifically designed for finding protocol vulnerabilities for router devices. IoTfuzzer [10] leverages the companion mobile apps of IoT devices to perform efficient black-box fuzzing. $FIRM-AFL$ [45] was presented as a high-throughput greybox fuzzer for firmware running a POSIX-compatible operating system through augmented process emulation. These emulation-based fuzzing approaches

Table 7: Comparison with $KARONTE$ and $SaTC$ on the same dataset.

| Vendor  | Samples | Alerts | $KARONTE$ [35] | KTP Rate | Time | Alerts | $SaTC$ [11] | KTP Rate | Time | Alerts | $EmTaint$ | KTP Rate | Time |
|---------|---------|--------|---------------|----------|------|--------|------------|----------|------|--------|------------|----------|------|
| NETGEAR | 17      | 36     | 23            | 63.9%    | 17:13| 1,901  | 537        | 28.2%    | 16:47| 849    | 849        | 100.0%   | 00:05|
| D-Link  | 9       | 24     | 15            | 62.5%    | 14:09| 32     | 22         | 68.8%    | 01:57| 299    | 234        | 78.3%    | 00:02|
| TP-Link | 16      | 2      | 2             | 100.0%   | 01:30| 7      | 2          | 28.6%    | 04:13| 73     | 73         | 100.0%   | 00:05|
| Tenda   | 7       | 12     | 6             | 50.0%    | 01:01| 122    | 122        | 84.7%    | 12:19| 362    | 362        | 100.0%   | 00:05|
| Total   | 49      | 74     | 46            | 62.2%    | 451:06| 2,084  | 683        | 32.8%    | 459:33| 1,583  | 1,518      | 95.9%    | 03:38|

For each tool, we report the total number of generated alerts, the number of true positives based on the definition in $KARONTE$ (# of KTP), the true-positive rate (KTP Rate), and the average analysis time for each sample (hh:mm). In the row labelled “Total”, we show the aggregated time to analyze all 49 samples in the column “Time”.

For each tool, we report the total number of generated alerts, the number of true positives based on the definition in $KARONTE$ (# of KTP), the true-positive rate (KTP Rate), and the average analysis time for each sample (hh:mm). In the row labelled “Total”, we show the aggregated time to analyze all 49 samples in the column “Time”.
rely on correctly executing the firmware in an emulator in the first place. However, accurate emulation is a non-trivial task in practice due to the diversity of embedded devices [28].

8 Conclusion

In this work, we propose a precise demand-driven flow-, context- and field-sensitive alias analysis approach. Based on this new approach, we also design a novel indirect call resolution scheme. Combined with sanitization rule checking, our solution discovers taint-style vulnerabilities by static taint analysis. We implemented our idea with a prototype called EmTaint and evaluated it against 35 real-world embedded firmware samples from six popular vendors. EmTaint discovered at least 192 bugs, including 41 n-day bugs and 151 0-day bugs. At least 115 CVE/PSV numbers have been allocated from a subset of the reported vulnerabilities at the time of writing. Compared to the state-of-the-art tools such as KARONTE and SaTC, EmTaint found significantly more bugs on the same dataset in less time.

References

[1] Claripy api. http://angr.io/api-doc/claripy.html. Last accessed: 2020-07-20.
[2] Common vulnerabilities and exposures. https://cve.mitre.org/. Last accessed: 2020-07-25.
[3] Exploit database of the website. https://www.exploit-db.com/. Last accessed: 2020-06-20.
[4] A python module for vex intermediate representation. https://github.com/angr/pyvex. Last accessed: 2020-07-23.
[5] Gogul Balakrishnan, Radu Gruiu, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In CC, 2005.
[6] Gogul Balakrishnan and Thomas Reps. Recovery of variables and heap structure in x86 executables. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2005.
[7] Gogul Balakrishnan and Thomas Reps. Divine: Discovering variables in executables. In VMCAI, 2007.
[8] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. TOPLAS, 2010.
[9] Daming D Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In NDSS, 2016.
[10] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiquang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering remote vulnerabilities in IoT devices using soft fuzzing. In NDSS, 2019.
[11] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In USENIX Security, 2021.
[12] Ben-Chung Cheng and Wen-Mei W Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In PLDI, 2000.
[13] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtain: detecting the taint-style vulnerability in embedded device firmware. In DSN, 2018.
[14] Shaoxin Cheng, Jun Yang, Jiajie Wang, Jinding Wang, and Fan Jiang. Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code. In IEEE TrustCom, 2011.
[15] Tat-Wai Chong and Peng-Sheng Chen. Accurate instruction-level alias analysis for arm executable code. In Advances in Intelligent Systems and Applications—Volume 2, pages 657–667. Springer, 2013.
[16] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. Science of Computer Programming, 40(2-3):171–188, 2001.
[17] Marco Cova, Viktoria Felmetger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In ACSAC, 2006.
[18] Bjorn De Sutter, K De Bosschere, Peter Keyngaert, and Bart Demeoen. On the static analysis of indirect control transfers in binaries. In PDPTA, 2000.
[19] Saumya Debray, Robert Muth, and Matthew Weippeert. Alias analysis of executable code. In ACM SIGPLAN-SIGACT POPL, 1998.
[20] Ivan Gogtovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In NDSS Workshop on BAR, 2018.
[21] Grammathec. A source code and binary code static analysis tool. https://www.grammathec.com/codesonar-sas-binary, 2021. Last accessed: 2021-06-08.
[22] Bolei Guo, Matthew J Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I August. Practical and accurate low-level pointer analysis. In CGO, 2005.
[23] Mahui Jiang, Yajin Zhou, Xiaopu Luo, Ruoyu Wang, Yiangu Liu, and Kui Ren. An empirical study on arm disassembly tools. In ISSTA, 2020.
[24] Karonte. The experimental dataset used by tool Karonte. https://github.com/ucsb-seclab/karonte#dataset, 2020. Last accessed: 2021-06-08.
[25] ReFirm Labs. The tool for analyzing, reverse engineering, and extracting firmware images. https://github.com/ReFirmLabs/binwalk, 2010. Last accessed: 2020-07-25.
[26] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In ACM CCS, 2019.
[27] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In ACSAC, 2000.
[28] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Franillon, and Davide Balzarotti. What you corrupt is not what you see is not what you execute. TOPLAS, 2010.
[29] Nicholas Nethercote and Julian Seward. Valgrind: a framework for lightweight dynamic binary instrumentation. ACM Sigplan notices, 42(6):89–100, 2007.
[30] Wei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiquang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In USENIX Security, 2014.
[31] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In USENIX Security, 2015.
Table 8: Taint sources and sinks.

| Taint Sources | recv, recvfrom, read, fread, fgets |
|---------------|-----------------------------------|
|               | BIO_read, BIO_gets, SSL_read, getenv |
| Taint Sinks   | strcpy, strncpy, memcpy, memmove |
|               | sprintf, sscanf, strcat, strncat |
|               | system, popen, execve |

Table 9: The summaries of library functions that propagate tainted data.

| Vendor | Vulnerability IDs | Alerts |
|--------|------------------|--------|
| Cisco  | CVE-2019-1663, CVE-2019-1652 | 2 |
| TRENDSnet | CVE-2019-13151, CVE-2019-13279 | 68 |
| D-Link | CVE-2018-17787, CVE-2019-9122 | 9 |
| Netgear | CVE-2017-6077, CVE-2017-6334 | 3 |
| TRENDnet | CVE-2018-18732, CVE-2020-13390 | 2 |
| Cisco  | CVE-2020-13391, CVE-2020-13392 | 2 |
| TRENDnet | CVE-2020-13393, CVE-2020-13394 | 2 |
| TEND | CVE-2018-14599, CVE-2018-14492 | 1 |
| TEND | CVE-2018-14557, CVE-2018-16333 | 36 |
| Netgear | CVE-2018-17806, CVE-2018-18707 | 2 |
| TEND | CVE-2018-18708, CVE-2018-18709 | 2 |
| TEND | CVE-2018-18727, CVE-2018-18730 | 2 |
| TP-Link | CVE-2018-16119, CVE-2017-13772 | 4 |
| Total  | 41 | 120 |

A Taint Source and Taint Sink

Table 8 shows all sources and sinks used in EmTaint.

B Library Function Summaries

Table 9 shows the summaries of 29 common functions from the Standard C Library.

C Vulnerability Discovery

Table 12 shows the alerts produced by EmTaint for the remaining 25 of 35 firmware samples. The results of the other 10 samples are shown in Table 3 in Section 5.1. Table 10 shows the n-day vulnerabilities found by EmTaint for 35 firmware samples. Table 11 shows the 0-day vulnerabilities found by EmTaint for 35 firmware samples.

D Indirect Call Resolution

Table 13 shows the results of indirect call resolution for the remaining 25 of 35 firmware samples. The results of the other
Table 11: Zero-day vulnerabilities found by EmTaint

| Vendor | Model | Vulnerability IDs |
|--------|-------|-------------------|
| Cisco  | RV320 | CVE-2020-3274, CVE-2020-3275, CVE-2020-3276, CVE-2020-3277, CVE-2020-3278, CVE-2020-3279, CVE-2020-3286, CVE-2020-3287, CVE-2020-3288, CVE-2020-3289, CVE-2020-3290, CVE-2020-3291, CVE-2020-3292, CVE-2020-3293, CVE-2020-3294, CVE-2020-3295, CVE-2020-3296, CVE-2021-1319, CVE-2021-1320, CVE-2021-1321, CVE-2021-1322, CVE-2021-1323, CVE-2021-1324, CVE-2021-1325, CVE-2021-1326, CVE-2021-1327, CVE-2021-1328, CVE-2021-1329, CVE-2021-1330, CVE-2021-1331, CVE-2021-1332, CVE-2021-1333, CVE-2021-1334, CVE-2021-1335, CVE-2021-1336, CVE-2021-1337, CVE-2021-1338, CVE-2021-1339, CVE-2021-1340, CVE-2021-1341, CVE-2021-1342, CVE-2021-1343, CVE-2021-1344, CVE-2021-1345, CVE-2021-1346, CVE-2021-1347, CVE-2021-1348, CVE-2021-1349, CVE-2021-1314, CVE-2021-1315, CVE-2021-1316, CVE-2021-1317, CVE-2021-1318. |
| Cisco  | RV130 | CVE-2020-3268, CVE-2020-3269, CVE-2021-1146, CVE-2021-1147, CVE-2021-1148, CVE-2021-1150, CVE-2021-1159, CVE-2021-1160, CVE-2021-1161, CVE-2021-1162, CVE-2021-1163, CVE-2021-1165, CVE-2021-1166, CVE-2021-1169, CVE-2021-1170, CVE-2021-1171, CVE-2021-1172, CVE-2021-1173, CVE-2021-1174, CVE-2021-1175, CVE-2021-1176, CVE-2021-1177, CVE-2021-1178, CVE-2021-1179, CVE-2021-1180, CVE-2021-1181, CVE-2021-1182, CVE-2021-1183, CVE-2021-1184, CVE-2021-1185, CVE-2021-1186, CVE-2021-1187, CVE-2021-1188, CVE-2021-1189, CVE-2021-1190, CVE-2021-1191, CVE-2021-1192, CVE-2021-1193, CVE-2021-1194, CVE-2021-1195, CVE-2021-1196, CVE-2021-1203, CVE-2021-1204. |
| D-Link | DIR-925 | CVE-2020-10213, CVE-2020-10215, CVE-2020-10217, CVE-2020-10216 |
| D-Link | DAP-1660 | 2 unassigned |
| TRENDnet | TEW632BRP | CVE-2020-10213, CVE-2020-10215, CVE-2020-10216 |
| TRENDnet | TEW827DRU | CVE-2020-14074, CVE-2020-14075, CVE-2020-14076, CVE-2020-14077, CVE-2020-14078, CVE-2020-14079, CVE-2020-14080, CVE-2020-14081, 14 unassigned |
| NETGEAR | R7800 | 1 unassigned |
| TP-Link | WR940NV4 | 2 unassigned |
| Tenda | AC5V | 4 unassigned |

Total: 10

151 vulnerabilities, 115 assigned with public exposure IDs

10 samples are shown in Table 5 in Section 5.2.
# Alerts produced by EmTaint for the remaining 25 firmware samples

| Vendor | ID   | Firmware Version | Arch | Binary | Size (KB) | Ana. Func | Tainted Sinks | Alerts | Time (s) |
|--------|------|------------------|------|--------|-----------|-----------|---------------|--------|----------|
| D-Link | 5    | DIR-645_1.03     | MIPS3| cgibin | 156       | 190       | 37            | 21     | 18.85    |
|        | 6    | DIR-890L_A1_1.03 | ARM32| cgibin | 151       | 305       | 49            | 21     | 42.26    |
|        | 7    | DIR-868L_A1_b04  | ARM32| cgibin | 151       | 368       | 43            | 8      | 38.63    |
|        | 8    | DIR-823G_A1_B03  | MIPS32| goahead | 1,525     | 1043      | 52            | 5      | 82.18    |
|        | 9    | DCS-5020L_A1_v1.15.12 | MIPS32| alphapid | 707       | 1,086     | 48            | 4      | 716.65   |
| NETGEAR | 14   | R6200v2_v1.0.3.12 | ARM32| httpd  | 1,259     | 852       | 257           | 52     | 123.00   |
|        | 15   | R6300v2_v1.0.4.18 | ARM32| httpd  | 1,294     | 864       | 254           | 24     | 132.70   |
|        | 16   | R6400_v1.0.1.46  | ARM32| httpd  | 1,482     | 1,123     | 401           | 36     | 148.74   |
|        | 17   | R7000_v1.0.1.36  | ARM32| httpd  | 1,762     | 1,320     | 464           | 36     | 273.97   |
|        | 18   | R7000P_v1.3.0.8  | ARM32| httpd  | 1,764     | 1,349     | 480           | 39     | 288.00   |
|        | 19   | R7300DST_v1.0.0.56 | ARM32| httpd  | 1,452     | 1,030     | 386           | 35     | 184.38   |
|        | 20   | R7500v2_v1.0.3.16 | ARM32| net-cgi | 603       | 1,473     | 226           | 75     | 110.89   |
|        | 21   | R7900_v1.0.1.26  | ARM32| httpd  | 1,499     | 1,080     | 425           | 37     | 209.03   |
|        | 22   | R8300_v1.0.2.106 | ARM32| httpd  | 1,490     | 1,043     | 393           | 37     | 152.15   |
|        | 23   | R9000_v1.0.2.40  | ARM32| net-cgi | 635       | 1,294     | 227           | 94     | 89.40    |
|        | 24   | DGN1000_v1.0.100.46 | MIPS32| setup.cgi | 324       | 732       | 88            | 56     | 30.55    |
|        | 25   | DGN2200_v1.0.0.50 | MIPS32| httpd  | 990       | 773       | 257           | 69     | 111.27   |
|        | 26   | AC1450_v1.0.0.36 | ARM32| httpd  | 1,230     | 835       | 246           | 49     | 92.70    |
|        | 27   | WNR3500Lv2_v1.2.0.46 | MIPS32| httpd  | 1,584     | 888       | 375           | 14     | 202.17   |
|        | 28   | XR500_v1.2.1.04  | ARM32| net-cgi | 536       | 1,327     | 152           | 35     | 81.70    |
|        | 29   | MR3020V1_en.3.17.2 | MIPS32| httpd  | 1,524     | 2,613     | 264           | 10     | 208.28   |
|        | 30   | WR1043NDV3_en.3.16.9 | MIPS32| httpd  | 1,900     | 3,629     | 254           | 32     | 373.21   |
|        | 31   | AC10V1.0RTL_v1.15.03.623 | MIPS32| httpd  | 2,011     | 1,187     | 166           | 82     | 358.26   |
|        | 32   | AC15V1.0BR_v1.15.03.18 | ARM32| app_data_center | 83       | 182       | 16            | 5      | 10.94    |
|        | 33   | WH450AV1BR_v1.0.0.18 | MIPS32| httpd  | 416       | 564       | 76            | 40     | 194.42   |

# Results of indirect call resolution for the remaining 25 firmware samples

| Model | All | Resolved | % of resolved |
|-------|-----|----------|---------------|
|       | I-Calls | I-Calls | I-Calls targets | I-Calls |
|       |        |         |               |         |            |
| D-Link | 5 | DIR-645 | 10 | 7 | 47 | 70.0% | 4.82 |
|        | 6 | DIR-890L | 5 | 4 | 24 | 80.0% | 3.60 |
|        | 7 | DIR-868L | 5 | 4 | 23 | 80.0% | 3.70 |
|        | 8 | DIR-823G | 43 | 25 | 399 | 58.1% | 54.25 |
|        | 9 | D-Link DCS-5020L | 94 | 73 | 478 | 76.6% | 410.10 |
| NETGEAR | 14 | R6200v2 | 3 | 2 | 410 | 66.6% | 27.20 |
|        | 15 | R6300v2 | 3 | 2 | 415 | 66.6% | 28.67 |
|        | 16 | R6400 | 3 | 2 | 486 | 66.6% | 38.06 |
|        | 17 | R6700 | 3 | 2 | 612 | 66.6% | 66.83 |
|        | 18 | R7000P | 3 | 2 | 617 | 66.6% | 73.21 |
|        | 19 | R7300DST | 3 | 2 | 492 | 66.6% | 37.44 |
|        | 20 | R7500v2 | 28 | 25 | 807 | 89.2% | 40.21 |
|        | 21 | R7900 | 3 | 2 | 484 | 66.6% | 37.72 |
|        | 22 | NETGEAR R8300 | 3 | 2 | 496 | 66.6% | 38.36 |
|        | 23 | NETGEAR R8900 | 52 | 47 | 581 | 90.3% | 36.87 |
|        | 24 | NETGEAR DGN1000 | 40 | 36 | 642 | 90.0% | 26.33 |
|        | 25 | NETGEAR DGN2200 | 3,206 | 3,204 | 476 | 99.9% | 130.91 |
|        | 26 | NETGEAR AC1450 | 3 | 2 | 414 | 66.6% | 30.46 |
|        | 27 | NETGEAR WNR3500Lv2 | 4,667 | 4,662 | 574 | 99.8% | 161.06 |
|        | 28 | NETGEAR XR500 | 29 | 13 | 629 | 44.8% | 32.27 |
| TP-Link | 30 | MR3020NV1 | 299 | 236 | 553 | 78.9% | 170.53 |
|        | 31 | WR1043NDV3 | 388 | 308 | 687 | 79.4% | 431.28 |
| Tenda | 32 | AC10V1.0RTL | 88 | 65 | 279 | 73.9% | 208.19 |
|        | 33 | AC15V1.0BR | 47 | 38 | 2 | 80.8% | 4.51 |
|        | 34 | WH450AV1BR | 2,063 | 2,043 | 269 | 99.0% | 118.16 |