
# Detecting Vulnerabilities in Linux-Based Embedded Firmware with SSE-Based On-Demand Alias Analysis

Kai Cheng
SIAT, CAS †
Sangfor Technologies Inc.
China
chengkai@sangfor.com.cn

Yaowen Zheng*
Nanyang Technological University
Singapore
yaowen.zheng@ntu.edu.sg

Tao Liu
The Pennsylvania State University
USA
tul459@psu.edu

Le Guan
University of Georgia
USA
leguan@uga.edu

Peng Liu
The Pennsylvania State University
USA
pxl20@psu.edu

Hong Li
IIE, CAS ‡
China
lihong@iie.ac.cn

Hongsong Zhu
IIE, CAS ‡
School of Cyber Security, UCAS §
China
zhuhongsong@iie.ac.cn

Kejiang Ye
SIAT, CAS †
China
kj.ye@siat.ac.cn

Limin Sun
IIE, CAS ‡
School of Cyber Security, UCAS §
China
sunlimin@iie.ac.cn

## ABSTRACT

Although the importance of using static taint analysis to detect taint-style vulnerabilities in Linux-based embedded firmware is widely recognized, existing approaches are plagued by following major limitations: (a) Existing works cannot properly handle indirect call on the path from attacker-controlled sources to security-sensitive sinks, resulting in lots of false negatives. (b) They employ heuristics to identify mediate taint source and it is not accurate enough, which leads to high false positives.

To address issues, we propose *EmTaint*, a novel static approach for accurate and fast detection of taint-style vulnerabilities in Linux-based embedded firmware. In *EmTaint*, we first design a structured symbolic expression-based (SSE-based) on-demand alias analysis technique. Based on it, we come up with indirect call resolution and accurate taint analysis scheme. Combined with sanitization rule checking, *EmTaint* can eventually discovers a large number of taint-style vulnerabilities accurately within a limited time. We evaluated *EmTaint* against 35 real-world embedded firmware samples from six popular vendors. The result shows *EmTaint* discovered at least 192 vulnerabilities, including 41 n-day vulnerabilities and 151 0-day vulnerabilities. At least 115 CVE/PSV numbers have been allocated from a subset of the reported vulnerabilities at the time of writing. Compared with state-of-the-art tools such as KARONTE and SaTC, *EmTaint* found significantly more vulnerabilities on the same dataset in less time.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**.

## KEYWORDS

On-demand alias analysis, Taint analysis, Embedded firmware

**ACM Reference Format:**
Kai Cheng, Yaowen Zheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, Kejiang Ye, and Limin Sun. 2023. Detecting Vulnerabilities in Linux-Based Embedded Firmware with SSE-Based On-Demand Alias Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598062

## 1 INTRODUCTION

With the emerging of the *Internet of Things* (IoT) technologies, embedded devices are widely deployed in our daily life. For fast development, vendors of devices prefer to customize Linux kernel and compose the embedded Linux-based firmware to manage and control the device. For example, almost all the mainstream wireless routers are equipped with Linux-based embedded firmware. However, due to lack of comprehensive security assessment from vendors, Linux-based embedded firmware suffers from a number of vulnerabilities without revealed, which has brought significant impact to cyberspace security. Among various vulnerability types, taint-style vulnerability is a typical vulnerability type where the user input can reach to a security-sensitive sink without proper check or sanitization [41]. Since Linux-based embedded devices

*Yaowen Zheng is the corresponding author
†Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
‡Institute of Information Engineering, Chinese Academy of Sciences
§School of Cyber Security, University of Chinese Academy of Sciences


*ISSTA '23, July 17–21, 2023, Seattle, WA, USA*

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598062

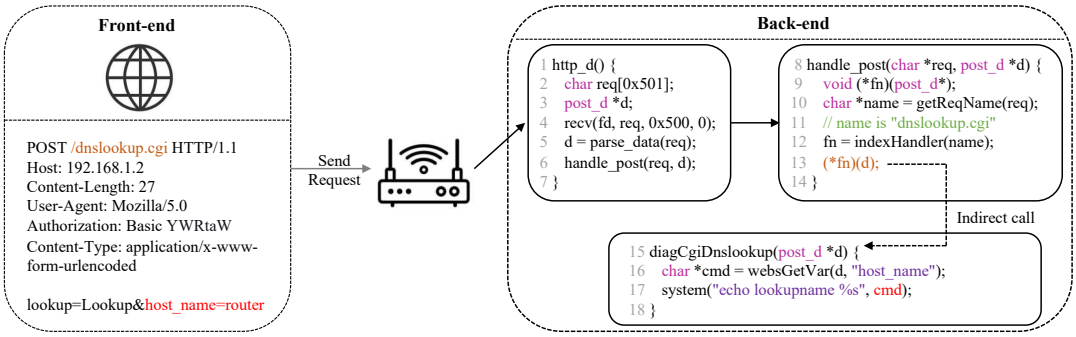


**Figure 1: Motivating example in the NETGEAR DGN2200 router.**

such as routers and web cameras frequently communicate with outside world, taint-style vulnerabilities are more likely to be triggered, so that we need to prioritize the detection for it.

Static taint analysis is an effective technique for finding taint-style vulnerabilities in embedded firmware, e.g, KARONTE [30] and SaTC [7]. It can be used to perform the entire analysis of firmware and does not require the emulation and real devices. In the static taint analysis, taint sources refer to library functions (e.g., recv, recvfrom) that receive the user input, and taint sinks refer to the security sensitive library function (e.g., strcpy, system) that can be exploited to cause memory corruption or command injection. The basic idea of taint analysis is to find data flow path from taint source to taint sink (we call it potential vulnerable path), and then perform the security check on the path to infer whether the vulnerability exists or not.

**Challenges.** However, effectively revealing potential vulnerable paths is a challenging problem for taint analysis on embedded binaries. The reason is twofold. On the one hand, in the back end of Linux-based firmware, most of the potential vulnerable paths involve indirect calls (see §4.3), some of which cannot be directly resolved. As the result, lots of potential vulnerable paths still remain unrevealed in the target binary. On the other hand, the frequent occurrence of variable alias also prevents data flow analysis techniques from finding implicit potential vulnerable paths. To mitigate these challenges, state-of-the-art works [7, 30] utilize heuristic methods to identify the mediate taint source as an alternative to the original taint source, so that it can shorten the length of potential vulnerable paths from user input to sink point, which reduce the probability of encountering indirect call and variable alias issues. Unfortunately, these approaches sometimes misidentify the mediate taint source in back-end binaries, which cause false positives of finding potential vulnerable paths. Meanwhile, indirect call resolution is still not considered in these works due to associated difficulties, so that many potential vulnerable paths are unrevealed, which produces many false negatives.

**Our solution.** In this paper, we aim at completely, accurately revealing potential vulnerable paths from taint source to taint sink by solving the issues brought by indirect calls and implicit data flow. Our insight is that accurate alias analysis can facilitate both indirect call resolution and implicit data flow analysis during the taint propagation. To this end, we propose a novel technique called *structured symbolic expression-based* (SSE-based) on-demand alias analysis. The technique overcomes the drawbacks of existing alias analysis techniques (VSA-based or symbolic-execution-based approaches) from two aspects. First, our proposed technique designs a new symbolic expression to represent alias information, which improve accuracy of alias analysis compared with existing works. Second, the technique only focuses on interesting variables (indirect call targets and tainted variables) to perform on-demand analysis, avoiding the holistic analysis of other irrelevant variables.

On the basis of SSE-based on-demand alias analysis, we propose *EmTaint*, which can effectively find taint-style vulnerabilities in Linux-based embedded firmware. The design of *EmTaint* is as follows. First, *EmTaint* extracts the target binary executable from embedded firmware and builds the basic program representation for further static analysis. Then, *EmTaint* performs SSE-based on-demand alias analysis, which can help find the alias relationship between the indirect call targets and function pointers. Based on it, *EmTaint* designs both indirect call resolution and accurate taint initialization and propagation scheme. Finally, *EmTaint* integrates sanitization rule checking, and can find taint-style vulnerabilities in Linux-based firmware both effectively and efficiently.

We have implemented the prototype of *EmTaint* with about 24,000 lines in Python. In *EmTaint*, the basic program representations are built by using IDA Pro [22], CLE [11], pyvex [29]. The proposed SSE-based on-demand alias analysis and indirect call resolution are developed based on Claripy [10]. We evaluated *EmTaint* with 35 real-world firmware samples. The result shows *EmTaint* discovered at least 192 vulnerabilities, including 41 n-day vulnerabilities and 151 0-day vulnerabilities, which takes an average of 3 minutes on each sample. We have reported 151 0-day vulnerabilities to the relevant manufacturers for responsible disclosure. 115 of vulnerabilities are confirmed by CVE/PSV at the time of writing. We also conducted a comparison with state-of-the-art works KARONTE [30] and SaTC [7]. The results of the comparison shown that *EmTaint* can find more vulnerabilities in less time. In addition, we used *EmTaint* in the 2021 DataCon competition for the vulnerability detection of IoT firmware, and finally won the first place, which demonstrates the efficiency of *EmTaint* in the real world.

**Contributions.** We make the following contributions:

- We proposed a novel alias analysis technique called SSE-based on-demand alias analysis, to handle the challenge of effectively revealing potential vulnerable path for taint analysis.
- We proposed *EmTaint*, a novel methodology for detection of taint-style vulnerabilities in Linux-based embedded firmware. The prototype was implemented with about 24,000 lines in Python.

- We evaluated *EmTaint* with extensive experiments. The result proves that *EmTaint* outperforms state-of-the-art works in detection of taint-style vulnerabilities in Linux-based firmware. Besides, *EmTaint* has discovered 151 0-day vulnerabilities (115 assigned with CVEs/PSVs) in 35 embedded firmware samples.
- We released the source code and experiment data of *EmTaint* at https://sites.google.com/view/emtaint/home for future research.

# 2 BACKGROUND AND MOTIVATION

## 2.1 Taint-Style Vulnerability Detection

Given an embedded binary, the standard taint analysis 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 sources and security-sensitive sinks. (3) Find a path where the taint is propagated from the source to the sink. (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 these steps.

## 2.2 Motivating Example

However, how to effectively reveal the complete path from taint source to sink (step 3) is a challenging problem. The main reason is that the service program in embedded devices often register callback functions to handle different kinds of user requests, which prevent taint propagation from source to sink. Only the user request is specified during the execution, the indirect call target can be assigned with one of callback functions. That means, the indirect call targets are still unknown when performing the static taint analysis and thus the taint analysis would get stuck at these points.

Figure 1 shows an example of how indirect calls prevent taint analysis techniques finding command injection vulnerability in NETGEAR DGN2200 router. First, the back-end receives the request from the front-end, and obtains URL `dnsloopup.cgi` through the `getReqName` function and stores it in variable `name` (line 10). Then, the back-end searches the function pointer in the global function table that is registered to handle `dnsloopup.cgi` (line 12) and finally calls function `diagCgiDnslookup` (line 13). The function `diagCgiDnslookup` contains a command injection vulnerability: the value of `cmd` comes from the user request (marked with red color in the front-end) without any sanitization check. An attacker can append additional commands to string 'router' to run arbitrary commands on the device. However, in the static taint analysis, $*fn$ (line 13) is unknown, so that taint analysis cannot reach to the `system` function and find this potential vulnerability.

Existing works, including SaTC [7] and KARONTE [30] use heuristics to infer the mediate taint source as representative of user input and thus shorten the distance from taint source to sink, which reduce the complexity of taint analysis. However, they still have limitations. First, the heuristics to identify the mediate taint source still produce lots of false positives and false negatives. Specifically, SaTC utilize shared interface keywords from both front-end web files and back-end binaries to identify mediate taint source. However, the keywords of some user input do not exist in the front-end web files. For example, in NETGEAR R7800 router with firmware version V1.0.2.58, the keyword *hidden_funjsq_username* cannot be found in the front-end web files, so that SaTC fails to identify it, but the value of it can cause a command injection vulnerability. Second, indirect calls issues still exist in shortened paths from mediate taint sources to sinks, which are not considered in existing works, leading to false negatives in revealing vulnerable paths.

## 2.3 Challenges and Key Idea

From the study of state-of-the-art works and the observation of the motivating example, we summarize two challenges for detecting taint-style vulnerabilities in Linux-based firmware. First, how to find a complete path from real input source (e.g., `recv` function) to the sink when indirect calls are involved on the path. Second, how to detect vulnerabilities efficiently while the path is deeper.

To address the above two challenges, we propose SSE-based on-demand alias analysis, which serves as a fundamental technique for indirect call resolution and accurate and efficient taint analysis. The SSE-based on-demand alias analysis can be explained from two aspects. First, we design a new symbolic expression called structured symbolic expression (SSE) inspired by access path [8]. However, access path targets source code, not the binary. SSE can accurately describe nested memory-access variables (e.g., `x.y.z`) to ensure the accuracy of alias analysis. Different from symbolic expression used in symbolic execution, SSE encodes both arithmetic operations and memory operations such as load and store. Thus, any forms of alias for given variables can be expressed accurately by means of SSE. Second, we perform on-demand analysis to ensure the efficiency of alias analysis. That means, only interesting variables, such as indirect call targets and tainted variables, need to be traced, avoiding the time-consuming holistic analysis. Based on SSE-based on-demand alias analysis, we take strategies to handle indirect calls and fulfil the accurate taint propagation. After that, we can reveal complete potential vulnerable paths from input sources to security-sensitive sinks effectively.

# 3 METHODOLOGY

## 3.1 Overview

*EmTaint* takes an embedded firmware image as input and reports potential taint-style vulnerabilities as output. As shown in Figure 2, *EmTaint* consists of four major components.

**Firmware Preprocessing.** The firmware preprocessing module decompresses and extracts binaries from firmware. Then, the module converts the code of binary into intermediate representation (IR). After that, it builds control flow graph (CFG) and a partial call graph (CG) that facilitate subsequent analysis.

**SSE-Based On-Demand Alias Analysis.** The alias analysis engine defines a new structured symbolic expression (SSE) to accurately represent the alias information, and designs alias update mechanisms to find aliases throughout the entire binary executable. To achieve on-demand analysis, it only computes aliases related to the variables of interest. This would avoid the overhead of calculating aliases for a large number of unrelated variables, so that the analysis can be applied to complex firmware binaries.

**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 indirect callsites and referenced function pointers (or function table pointers). The mechanism helps

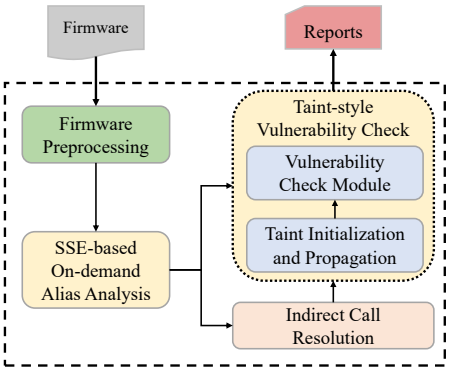


**Figure 2: The overview of *EmTaint***

subsequent taint analysis cross more function boundaries, and improve the discovery capability of taint-style vulnerabilities.

**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., `recv` function). Then, it captures taint propagation via the on-demand alias analysis module and taint propagation operations. Finally, at security-sensitive sinks (e.g., the `system` function), if the corresponding variable is tainted and unconstrained, we mark it as a potential vulnerability. Since the high-level idea of taint-style vulnerability detection is consistent with [3, 7, 30], so we include the detailed process description in [37].

## 3.2 SSE-Based On-Demand Alias Analysis

In this section, we first start with the definition of structured symbolic expressions (SSE). Then, we model the on-demand alias analysis problem with SSE. Based on it, we described how to perform updates to SSE on each instruction.

**Table 1: Recursive definition of SSE.**

| | | |
|---|---|---|
| $expr$ | ::= | $expr \lozenge_b expr \mid \lozenge_u expr \mid var$ |
| $\lozenge_b$ | ::= | $+, -, *, /, \ll, \gg, \ldots$ |
| $\lozenge_u$ | ::= | $\sim, !, \ldots$ |
| $var$ | ::= | $\tau_{reg} \mid \tau_{mem} \mid \tau_{val}$ |
| $\tau_{reg}$ | ::= | $r_i$ |
| $\tau_{val}$ | ::= | $\{Integer\}$ |
| $\tau_{mem}$ | ::= | $load(expr) \mid store(expr)$ |

*3.2.1 Definition of SSE.* SSE is a new notation that uses abstract memory model to represent aliases of a variable by encoding the nested memory-access information at relevant program points. In Table 1, we recursively define an SSE expression. Note that since our implementation is based on 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 $\tau_{reg}$), a primitive immediate value (denoted as $\tau_{val}$), or a memory access (denoted as $\tau_{mem}$). 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.

**An Intuitive Example.** We use an intuitive example to explain how we leverage SSE to find aliases for a given pointer. In the code snippet in Figure 3, there are four instructions. Our goal is to find all aliases of `R1` in line 3, which is known to be a pointer. First, we initialize the alias set with `R1` in line 3′. By looking backward, we find a definition of `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 4. 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 4′ to the alias set. By doing so back and forth iteratively along the define-use and use-define chain, we can eventually reach a fixed point of the alias set for the given pointer. In particular, we only need to calculate aliases related to variables of interest and avoid analyzing other extraneous variables (e.g., the variable at line 2) to improve speed. The full SSE update rules on each instruction and the algorithm for inter-procedural analyses are explained in the following sections.

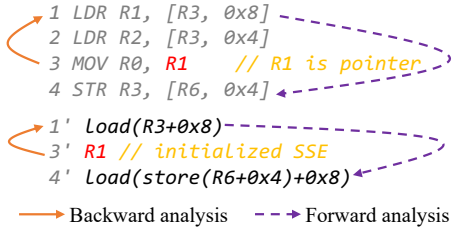


**Figure 3: Example to illustrate SSE-based alias analysis**

*3.2.2 On-Demand Alias Analysis with SSE.* Starting with the inter-procedural CFG and a query (e.g., a pointer $p$), our on-demand alias analysis algorithm involves two main aspects: (Aspect 1) how the alias analysis of one particular function works; (Aspect 2) how the termination of the analysis of one function triggers the analysis of another function to get started.

**Aspect 1 of our algorithm.** We first define a graph called *alias analysis graph* (AAG) to formally model the update and propagation behavior of SSE-based analysis within a particular function.

Definition 1 (AAG). *AAG is a directed, attributed graph G (N, E, S, Ftran, Btran, Fprop, Bprop), where N is a set of nodes $\{n_1, n_2, ..., n_k\}$ with each node $n_i$ representing an instruction, E is a set of directed edges representing the control flow transfer relation between nodes, each node has four separate sets of SSEs serving as node attributes, and S is the union of all the SSE sets on each node. Finally, Ftran and Btran are the two sets of transfer functions that update the SSE sets on each node during forward analysis and backward analysis, respectively. Fprop and Bprop are the two sets of functions that propagate SSEs along the direction and against the direction of each edge, respectively.*

Since the alias analysis is performed in both forward and backward directions, $S$ in AAG records SSE information of each node for both directions. The concrete definition is as follows.

- $S_n$: For a node $n$, $S_n$ is consisted of four SSE sets: $fin$, $fout$, $bin$, and $bout$. $fin$ is only used in forward analysis, while $bin$ is only used in backward analysis.
- $Ftran_n$: For a node $n$, it takes $fin$ of node $n$ and the instruction on node $n$ as inputs and updates two SSE sets $fout$ and $bout$.
- $Btran_n$: For a node $n$, it takes $bin$ of node $n$ and the instruction on node $n$ as inputs and updates two sets $fout$ and $bout$.

- $Fprop_{n\to m}$: For an edge $n \to m$, it propagates the SSEs held in $fout$ of node $n$ to the $fin$ set of node $m$.
- $Bprop_{n\to m}$: For an edge $n \to m$, it propagates the SSEs held in $bout$ of node $m$ to the $bin$ of node $n$.

Definition 2 (AAG-Q). *AAG-Q is an on-demand alias analysis method based on AAG. Given the AAG of a particular function $f$ and a pointer $p$ at node $m$ as query, AAG-Q runs the following algorithm to output $p$'s potential aliases on every node in the function's AAG:*

1) Initialize the $fin$ set of node $m$ as $\{p\}$.

2) Run a multi-iteration process until the following termination condition is met: the $fin$ sets and the $bin$ sets on all nodes no longer grow with an additional iteration. It should be noticed that a *fixed point* will be reached when the termination condition is met.

3) Each iteration consists of two passes: the forward analysis pass and the backward analysis pass.

4) During each iteration, the forward analysis pass works as follows:

- First, each node is accessed through reverse-postorder traversal on graph AAG of function $f$.
- When node $n$ is visited, if it is neither a call site node nor the function's start/return node, perform the following operations:
  $fout_n, bout_n = Ftran_n(fin_n, n)$
  $Fprop_{n\to s}: \quad fin_s = fin_s \cup fout_n, s \in succ_n$
  $Bprop_{r\to n}: \quad bin_r = bin_r \cup bout_n, r \in pred_n$,
  where $s \in succ_n$ denotes that $s$ is a successor node of $n$ in the graph AAG, and $r \in pred_n$ denotes that $r$ is a predecessor node of $n$.
- In case the current node is a call-site node, the modifications and references summary of the callee function will be firstly calculated using the technique proposed in [8]. Then a special transfer function will take the calculated summary and $fin$ of the call-site node as inputs and updates two sets $fout$ and $bout$ of the call-site node. After the updating is completed, the SSEs in $fout$ and $bout$ are propagated with $Fprop$ and $Bprop$ described in aforementioned operations.
- In case the current node is a start node, it performs the aforementioned operations except for $Bprop$.
- In case the current node is a return node, it performs the aforementioned operations, except that function $Fprop$ is implemented as follows:
  $Fprop_{n\to s}: \quad fin_s = fin_s \cup Choose(fout_n), s \in retsites_f$,
  where $Choose()$ operation selects the SSEs related to the return register (e.g., $R0$ in ARM) of the function $f$ and the SSEs related to global pointers, and $s \in retsites_f$ denotes that $s$ is a return-site node from function $f$ to caller.

5) During each iteration, the backward analysis pass works as follows:

- First, each node is accessed through postorder traversal on graph AAG of function $f$.
- When node $n$ is visited, if it is neither a call-site node nor the function's start/return node, perform the following operations:
  $fout_n, bout_n = Btran_n(bin_n, n)$
  $Fprop_{n\to s}: \quad fin_s = fin_s \cup fout_n, s \in succ_n$
  $Bprop_{r\to n}: \quad bin_r = bin_r \cup bout_n, r \in pred_n$
- In case the current node is a call-site node, the modifications and references summary of the callee function will be firstly calculated using the technique proposed in [8]. Then a special transfer function will take the calculated summary and $bin$ of the call-site node as inputs and updates two sets $fout$ and $bout$ of the call-site node. After the updating is completed and the SSEs in $fout$ and $bout$ are propagated with $Fprop$ and $Bprop$ described in aforementioned operations.
- In case the current node is a start node, it performs the aforementioned operations, except that function $Bprop$ is implemented as follows:
  $Bprop_{c\to n}: \quad bin_p = bin_p \cup Choose(bout_n), \begin{cases} c \in callsites_f \\ p \in pred_c \end{cases}$,
  where $Choose()$ operation selects the SSEs related to the argument registers (e.g., $R0$ in ARM) of the function $f$ and the SSEs related to global pointers, $c \in callsites_f$ denotes that $c$ is a call-site node for calling function $f$.
- In case the current node is a return node, it performs the aforementioned operations except for $Fprop$.

Note: the update rules with function summary is explained in detail in [37].

**Aspect 2 of our algorithm.** Our algorithm conducts inter-procedural on-demand alias analysis, but this does not mean that our algorithm will actively work on every function (in the target program) all the time. In fact, many functions will stay "dormant" until being triggered, and the alias analysis of a function will be triggered when any of the following conditions is met. (a) The query will automatically trigger the alias analysis of the function which uses pointer $p$. The function is the first one which will be analyzed by our algorithm. At this moment, all the other functions are actually "dormant". (b) When the alias analysis of a function $A$ terminates, if $A$ has a caller function, the analysis of the caller function will be triggered. When condition (a) is met, Aspect 1 of our algorithm describes all the operations performed by our algorithm. However, when condition (b) is met, our algorithm does the following beyond Aspect 1. First, it checks whether the $fin$ set of the return-site node and the $bin$ set of the predecessor node of the call-site node in the caller function's AAG-Q have changed. Then, if it changes, the caller function is analyzed as Aspect 1. Otherwise, the caller function goes back to "dormant".

*3.2.3 Updating Rule within Transfer Function.* In the above section, we introduce the iterative algorithm of inter-procedural on-demand alias analysis, which involves the transfer functions ($Ftran$ and $Btran$) that update the SSE sets on each normal node, In this section, we illustrate the corresponding update rules in detail. At each normal node, the rules of $Ftran_n$ and $Btran_n$ is shown in Table 2.

**(1) Update rules of Ftran.** The rules of $Ftran$ are listed in entries 1-7 in Table 2. In the table, each rule is represented as $statement \xrightarrow[c]{rn} expr.replace(rn, rj)$. Given the instruction of node, and the tracking SSE $expr$ from $fin$ of node, we first find the matched rule where $statement$ can represent the instruction. After finding the exact rule, we then check if $c$ is true and $rn$ exists in $expr$. If satisfied, new SSE should be generated where $rn$ is replaced with $rj$. Note that this process should be conducted over all the live variables in the current SSE. To explain the rules of $Ftran$ more clearly, we give an example here. The given instruction is $d = v$,

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

| **SSE update with define-use chain** | |
|---|---|
| (1)$ri = rj \xrightarrow{rj} expr.replace\,(rj, ri)$ | (2)$ri = Binop\,(rn, rm) \xrightarrow{rn \diamondsuit_b rm} expr.replace\,(rn \diamondsuit_b rm, ri)$ |
| (3)$ri = ITE\,(rj, rn, rm) \xrightarrow{rn} expr.replace\,(rn, ri)^{\S}$ | (4)$ri = ITE\,(rj, rn, rm) \xrightarrow{rm} expr.replace\,(rm, ri)^{\S}$ |
| (5)$ri = Load\,(rj) \xrightarrow{load(rj)} expr.replace\,(load\,(rj), ri)$ | (6)$Store\,(ri) = rj \xrightarrow{rj} expr.replace\,(rj, store\,(ri))$ |
| (7)$ri = Load\,(rj) \xrightarrow{store(rj)} expr.replace\,(store\,(rj), ri)$ | |
| **SSE update following use-define chain** | |
| (8)$ri = rj \xrightarrow{ri} expr.replace\,(ri, rj)$ | (9)$ri = Binop\,(rn, rm) \xrightarrow{ri} expr.replace\,(ri, rn \diamondsuit_b tm)$ |
| (10)$ri = ITE\,(rj, rn, rm) \xrightarrow{ri} expr.replace\,(ri, rn)^{\S}$ | (11)$ri = ITE\,(rj, rn, rm) \xrightarrow{ri} expr.replace\,(ri, rm)^{\S}$ |
| (12)$ri = Load\,(rj) \xrightarrow{ri} expr.replace\,(ri, load\,(rj))$ | (13)$Store\,(ri) = rj \xrightarrow[*]{load(ri)} expr.replace\,(load\,(ri), rj)$ |
| **SSE kills** | |
| (14)$ri = rj \xrightarrow{ri} expr.kill\,()$ | (15)$Store\,(ri) = rj \xrightarrow[+]{store(ri)\ or\ load(ri)} expr.kill()$ |

§: The statement $ri = ITE(rj, rn, rm)$ denotes that if $rj$ is true, $ri = rn$, otherwise, $ri = rm$. We ignore conditional judgments in the update rules.
*: Existing $load(ri)$ in $expr$ occurs after the newly encountered $store(ri)$ statement.
+: Existing $store(ri)/load(ri)$ in $expr$ occurs before the newly encountered $store(ri)$ statement.

and the current SSE is $v + o$. Thus, we identify the rule 1 in Table 2 as the matched one and a new SSE $d + o$ is generated by replacing the variable $v$ in the original SSE with the variable $d$. The new SSE generated by rules 1-7 is saved to set $fout$ of node. In particular, the new SSE generated by rule 6 is also saved in set $bout$ of node and used to find aliases of address $ri$ in backward iterative analysis.

**(2) Update rules of Btran.** The rules of *Btran* are listed in entries 1-6 (except for entry 7) and 8-13 in Table 2. Similarly, each rule is represented as $statement \xrightarrow[c]{rn} expr.replace(rn, rj)$. Given the instruction of the node, and the tracking SSE $expr$ from $bin$ of the node, we first find the matched rule where $statement$ can represent the instruction. After finding the exact rule, we then check if $c$ is true and $rn$ exists in $expr$. If satisfied, new SSE should be generated where $rn$ is replaced with $rj$. We also give an example here. The given instruction is $v = u$ and the current SSE is $load(v)$. Thus, we identify the rule 8 in Table 2 as the matched one, and a new SSE $load(u)$ is generated by replacing the variable $v$ in the original SSE with the variable $u$.

Unlike *Ftran*, which only checks whether the use variable (entries 1-7 in Table 2) exists in the tracking SSE $expr$, *Btran* needs to check use (entries 1-6 in Table 2) and definition variable (entries 8-13 in Table 2), both of which generate aliasing relationships. It is worth noting that in backward analysis, the new SSE updated by entries 1-5 is saved in set $fout$ of node, that is, it can only propagate forward, and the new SSE updated by entries 6 and 8-13 is saved in both set $fout$ and $bout$ of node, that is, it can propagate forward as well as backward.

**(3) Conditions for terminating an SSE.** Each variable has its own live scope. If a register assignment statement is encountered during forward or backward analysis, the liveness of the corresponding register (i.e., $\tau_{reg}$) is killed and the entire SSE associated with this register is terminated. This corresponds to rule 14 in Table 2. Similarly, if a *store* statement is encountered during forward analysis, the liveness of the corresponding memory access (i.e., $\tau_{mem}$) is killed and the entire SSE associated with this variable is terminated. This corresponds to rule 15 in Table 2.

*3.2.4 Handling Data-Flow Cycles and Recursive Data Structure.* The SSE describing memory access variables is a storeless mode [8] and represents an unbounded value domain, which may cause our algorithm to fail to reach a fixed point. To solve this problem, *EmTaint* utilizes k-limiting [16] method to limit the length of SSE, where $k$ is set to 5. With k-limiting, the input sets $fin$ and $bin$ of our algorithm guarantee convergence and our algorithm guarantees to reach a fixed point. In addition to the k-limiting method, *EmTaint* also uses access-path abstraction to abstract data-flow cycles and recursive data structure.

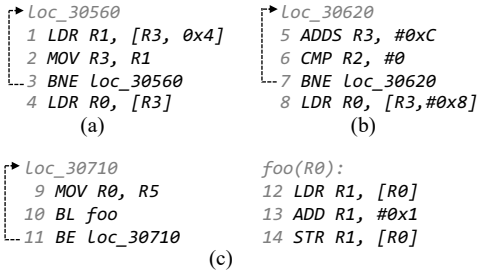


**Figure 4: Examples that cause SSE to be infinitely long**

We summarize three cases that lead to infinite SSE in our on-demand alias analysis. (1) The access of a recursive data structure in a loop. For example, in Figure 4(a), register $R3$ points to a recursive data structure. Querying the pointer $R3$ will obtain the aliases SSEs as `load(R3+0x4)`, `load(load(R3+0x4)+0x4)` and so on. (2) The access of incremented pointer in a loop, where the pointer value is incremented directly each loop iteration (e.g., register $R3$ in Figure 4(b) is incremented by `0xC` in each iteration). (3) The access of incremented pointer in a loop, the pointer value is incremented indirectly by calling a function. For example, $foo$ in Figure 4(c) is called through a loop, where the callee $foo$ increases the value pointed to by the pointer parameter $R0$ by `0x1` in each loop iteration. These three cases have the same feature: cyclic data dependence that exists in backward iterative update.

To this end, we take steps to prevent the generation of infinite SSEs in these three cases. For the case (1), when an alias SSE is updated in the same load instruction twice in backward analysis, we abstract the SSE as recursive expression $Rload(base + off)$, which represents $load(base + off)$, $load(load(base + off) + off)$ and so on. For the case (2) and case (3), when an alias SSE is updated in the same add instruction like '`ADD R3, cons`' (or function summary like $store(R0) = load(R0) + cons$) twice in backward analysis, we abstract the SSE as increasing expression $base + i * cons$, where $i$ represents the number of iterations of the loop and $cons$ represents the constant increase in the value of pointer $base$ each iteration.

*3.2.5 Termination, Completeness, Soundness, and Complexity.* The proposed on-demand alias analysis AAG-Q is inspired by access-path [8] and $k$-limiting approaches [16], which help bound the length of SSE. The main difference is that existing approaches only iteratively track unidirectional data dependencies but we perform *bidirectional* data dependencies, obtaining the forward analysis output $fout$ and the backward output $bout$, respectively. Therefore, the termination condition of our algorithm is similar to that of standard data-flow analysis with access-path and $k$-limiting. In particular, our algorithm guarantees to reach a fixed point.

Based on Meyer's article about completeness and soundness [5], we unambiguously define completeness and soundness in the setting of alias analysis as follows. AAG-Q is sound if AAG-Q says p1 and p2 are aliases, then p1 and p2 are indeed aliases. AAG-Q is complete if p1 and p2 are aliases, then AAG-Q will say p1 and p2 are aliases. Under this definition, AAG-Q is unsound because $k$-limiting is adopted. Specifically, by bounding the length of SSEs, AAG-Q may wrongly report aliases which are not. Based on our evaluation, among 166 reported aliases, 158 were true positives (see Section 4.2). In addition, AAG-Q may miss aliases when 1) the disembler makes mistakes, or 2) the algorithm reaches 15 iterations for a function. Therefore, our analysis is incomplete. The first factor is unavoidable due to the fundamental problem with any binary analysis method. That is, disambiguating between references and literal values in a binary is undecidable in general [39]. The second factor is imposed by ourselves due to performance considerations. Specifically, to strike the balance between soundness and efficiency, we made a compromise by limiting the number of iterations to 15 for each function in our on-demand alias analysis within a function (Section 3.2.2). During our evaluation, we found that the execution time could become unacceptable for some complex programs (e.g., with too many branches inside a loop). The number 15 was chosen mainly based on our empirical study which shows that increasing the number of iterations over 15 did not result in a better result in resolving indirect calls and discovering vulnerabilities. Even with this limit, we successfully found 158 aliases among 166 confirmed aliases (see Section 4.2). Since this is a tunable parameter, analysts can always select one that works best for the target program based on the time budget and the soundness requirement.

The complexity of our algorithm can be measured with the following parameters: $\mathbb{N}$ which denotes the total number of nodes in graph AAG of all functions, $\mathbb{V}_f$ which denotes the number of variables in a single function, $\mathbb{F}$ which denotes the total number of fields in all structures, and $k$ which denotes the value of $k$-limiting. The worst-case complexity can be expressed as $O(|\mathbb{N}||\mathbb{V}_f|^2|\mathbb{F}|^{2k})$ for both time and space. The complexity of our algorithm is consistent with inter-procedural pointer analysis [8], which is considered acceptable in practice.

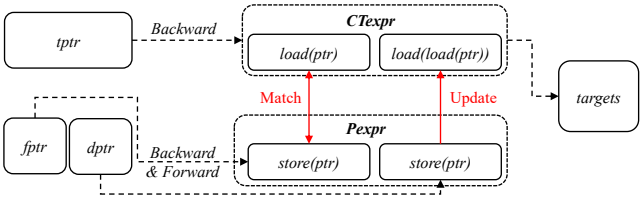


**Figure 5: Overview of indirect call resolution.**

## 3.3 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 through the disassembled code. *EmTaint* treats all branch instructions with a register as 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 6).

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. Here the address-taken functions stand for functions that have function pointers referring to them. 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 $fptr$ 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 $dptr$ and show an example in line 7. In this example, an entry in the table has two fields – a string pointer to the index name and the actual function pointer. For the target of indirect call, we denote it as $tptr$ and show an example in line 5 and 13.

```
1  void (*fun_ptr)() = &fun; //address-taken function
2  fptr = fun_ptr;
3  x.y.z = fptr;
4  fp1 = x.y.z;
5  call fp1;  //indirect call
6
7  dptr = 0x92C44;  //function table address
8  while ( strcmp(*dptr, name) ){
9    dptr += 0x8;
10   ...
11 }
12 fp2 = *(dptr+0x4);
13 call fp2;  //indirect call
```

**Listing 1: Indirect call resolution example.**

Finally, *EmTaint* performs SSE-based alias analysis and find dependence between indirect call targets $tprt$ and the original references $fptr/dptr$. Figure 5 illustrates the process of indirect call resolution. *EmTaint* back tracks the $tprt$ to find its all aliases. Meanwhile, *EmTaint* finds the aliases of $fptr$ and $dptr$ through both

**Table 3: Alerts produced by *EmTaint* for 35 samples**

| Vendor | ID | Firmware Version | Arch | Binary | Size (KB) | Ana. Func | Tainted Sinks | Alerts | Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| **Cisco (2)** | 1 | RV320_v1.4.2.20 | MIPS64 | ssi.cgi | 1,820 | 1,567 | 1450 | 335 | 634.64 |
| | 2 | RV130_v1.0.3.44 | ARM32 | httpd | 612 | 796 | 402 | 150 | 60.96 |
| **D-Link (7)** | 3 | DIR-825_B_2.10 | MIPS32 | httpd | 531 | 447 | 198 | 36 | 95.02 |
| | 4 | DAP-1860_A1_B03 | MIPS32 | uhttpd | 1,129 | 1,030 | 106 | 12 | 97.02 |
| **TRENDnet (2)** | 10 | TEW632BRP_1.010B32 | MIPS32 | httpd | 314 | 315 | 149 | 26 | 40.08 |
| | 11 | TEW827DRU_2.04B03 | MIPS32 | ssi | 998 | 622 | 289 | 142 | 58.20 |
| **NETGEAR (17)** | 12 | R7800_v1.0.2.32 | ARM32 | net-cgi | 542 | 1,286 | 291 | 119 | 88.91 |
| | 13 | R8000_v1.0.4.4 | ARM32 | httpd | 1,508 | 1,088 | 428 | 36 | 167.48 |
| **TP-Link (3)** | 29 | WR940NV4_us.3.16.9 | MIPS32 | httpd | 1,691 | 3,481 | 225 | 31 | 343.16 |
| **Tenda (4)** | 32 | AC9V3.0_v15.03.06.42 | MIPS32 | httpd | 2,039 | 1,201 | 172 | 84 | 433.62 |
| **Total (35)**§ | - | - | - | - | - | 38,983 | 9,346 | 1,887 | 179.81† |

§: The row labelled "Total" shows aggregated results for 35 samples. Due to page limit, we only list 10 representative samples in the table. The other 25 samples can be found in Table 4 in [37].
†: The average execution time per sample.

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 $tprt$, and matches and updates them. We denote the alias SSEs of the $tprt$ as $CTexpr$ (call target expression), and the alias SSEs of the $fptr$ or $dptr$ as $Pexpr$ (pointer expression). Next, we explain how to analyze $CTexpr$ and $Pexpr$ and infer indirect call targets associated with function table.

In the simplest form, a $CTexpr$ and a $Pexpr$ match directly. *EmTaint* can calculate the call target directly from $CTexpr$. This often indicates an indirect call referenced by $fptr$. If the $CTexpr$ can be represented by $load(dptr+i*stride)$, where the $dptr$ is the address of a function table, the $stride$ is a constant, and $i$ is an index, it strongly indicates a function table reference. In this case, *EmTaint* attempts to read values from addresses $dptr + i * stride$, where the index $i$ 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, *EmTaint* adds it to the set of indirect call targets.

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

## 4 IMPLEMENTATION AND EVALUATION

*EmTaint* is implemented with about 24,000 lines in Python (not including the open source code), and the implementation details can be found in [37]. In the evaluation, we aim to answer the following four questions. (1) How effective is it in uncovering real-world taint-style vulnerabilities in embedded firmware (§4.1)? (2) How accurate is the SSE-based on-demand alias analysis algorithm(§4.2)? (3) How effective is it in indirect call resolution with the support of SSE-based alias analysis? To what extent does the indirect call resolution play a role in improving vulnerability discovery (§4.3)? (4) How effective is it compared with the state-of-the-art techniques (§4.4)?

**Table 4: True positive evaluation by random sampling**

| ID | Model | Alerts | # of Samples | # of TP |
|---|---|---|---|---|
| 1 | Cisco RV320 | 335 | 34 | 32 |
| 2 | Cisco RV130 | 150 | 15 | 15 |
| 3 | D-Link DIR-825 | 36 | 4 | 3 |
| 4 | D-Link DAP-1860 | 12 | 2 | 2 |
| 10 | TRENDnet TEW632BRP | 26 | 3 | 2 |
| 11 | TRENDnet TEW827DRU | 142 | 14 | 7 |
| 12 | NETGEAR R7800 | 119 | 12 | 10 |
| 13 | NETGEAR R8000 | 36 | 4 | 4 |
| 29 | TP-Link WR940NV4 | 31 | 4 | 4 |
| 32 | Tenda AC9 | 84 | 8 | 7 |
| Total | 10 | 971 | 100 | 86 |

**Experiment Setup.** Our evaluation dataset consists of 35 different firmware samples from six major emebedded system vendors: Cisco, D-Link, NETGEAR, TRENDnet, TP-Link and Tenda. The majority of the firmware samples (31 samples) are selected from the dataset of the baseline KARONTE [30]. The remaining four samples are manually downloaded from the official websites of Cisco and TRENDnet. The summarized information is listed in Table 3, where `Size` denotes the size of the binary file we test (e.g., httpd). We can see from Table 3 that samples covers three architectures including ARM32, MIPS32 and MIPS64, which are the mainstream architectures used in Linux-based embedded devices. All the experiments were conducted on a Ubuntu 18.04.4 LTS OS running on PC with a 8-core Intel Core i7-8550U CPU and 24 GB RAM.

### 4.1 Vulnerability Discovery

Table 3 shows the results that *EmTaint* has found 9,346 different sinks where tainted data can reach to them. 1,887 of them were identified as alerts because no security sanitization was detected. The analysis time for each sample is about 180 seconds on average.

**True Positive Evaluation.** Given a large amount of raised alerts produced by *EmTaint*, it is important to study how many of them indicate true positive. However, the overall evaluation of true-positive

**Table 5: Comparison with *VSA* on the same dataset**

| Dataset | MAY-ALIAS | NO-ALIAS | *VSA* TP | *VSA* TN | *VSA* TPR | *VSA* ACC | *EmTaint* TP | *EmTaint* TN | *EmTaint* TPR | *EmTaint* ACC |
|---|---|---|---|---|---|---|---|---|---|---|
| **basic** | 67 | 27 | 46 | 26 | 68.7% | 76.6% | 61 | 27 | 91.0% | 93.6.0% |
| **flow** | 26 | 29 | 21 | 29 | 80.8% | 90.9% | 26 | 29 | 100.0% | 100.0% |
| **context** | 73 | 43 | 49 | 43 | 67.1% | 79.3% | 71 | 35 | 97.3% | 91.4% |
| **Total** | 166 | 99 | 116 | 98 | 69.9% | 80.8% | 158 | 91 | 95.2% | 94.0% |

is labor intensive, because it requires reverse-engineering and manual construction of a proof-of-concept (PoC) on real devices to validate whether they are real bugs or not. Therefore, we randomly selected a part of samples to estimate true positive rate. Specifically, we selected 10 physical devices and randomly sample 100 alerts out of 971 from these devices. Then, we identify an alert that refers to a true positive if (1) it matches a known vulnerability or (2) can be verified by successfully crafting a PoC on the real device. Table 4 shows that 86 out of 100 alerts refer to true positive, in which 13 true positives verified by known vulnerabilities and others verified via successfully constructed PoCs. This proves the high true positive rate of *EmTaint*. For the rest of false-positive alerts, we attribute them to inaccurate indirect call resolution, failures of finding security checks, fake alias relationship and taint sources.

**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 real devices that we have access to. To confirm 0-day vulnerabilities, we tried to craft PoCs against the latest firmware versions of real devices. The rule of identifying a vulnerability from multiple alerts is as follows: alerts triggered by parameters from POST request of the same URL interface at different sinks should be identified as one vulnerability. The rule is inspired by our observation of n-day vulnerabilities found by *EmTaint*. Specifically, we identify 41 n-day vulnerabilities by searching records from exploit-db [18] or MITRE CVE [13], which correspond to 120 alerts. Table 3 in [37] list the distribution of 41 n-day vulnerabilities among the public exposure IDs. Note that one CVE ID may correspond to multiple alerts. For example, CVE-2019-13278 refers to a unique command injection vulnerability, which can be triggered at 33 different sinks in the target binary. Finally, we have confirmed a total of 151 0-day vulnerabilities from 512 alerts. We reported the 151 vulnerabilities to manufacturers, all of which have been fixed. The discovered 0-day vulnerability include 38 command injection vulnerabilities and 113 buffer overflow vulnerabilities, 115 of which have been assigned with CVE/PSV numbers. The vulnerability information including relevant CVEs/PSVs is summarized in Table 5 in [37]. Since some samples from our dataset are in older version, lots of alerts generated by *EmTaint* have been fixed in the latest version. We infer that these vulnerabilities were found by the vendor themselves, and 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 vulnerabilities were fixed by replacing `strcpy` with `strncpy`.

In summary, *EmTaint* has found 151 0-day vulnerabilities (115 CVE/PSV) and 41 n-day vulnerabilities on our evaluation dataset. Meanwhile, *EmTaint* can find vulnerabilities with high true-positive (86%).

## 4.2 Effectiveness of SSE-Based Alias Analysis

To evaluate the effectiveness of SSE-based alias analysis, we select VSA [4], the most advanced alias analysis technique for binaries as the baseline. The selected test dataset, PTABEN [28], is a benchmark used in SUPA [36], which is the state-of-the-art demand-driven alias analysis technique for source code. We choose three test sets of C program from PTABEN, including basic test set, context-sensitive test set and flow-sensitive test set. The three test sets we selected covered all types of pointers. The rest of test sets are written in other languages that are not relevant to the scope of our analysis. We use Angr v9.0.5811 [2], which implemented VSA, to carry out the comparison experiment.

Table 5 shown the results. The *MAYALIAS(*p, *q)* indicates that pointers $p$ and $q$ are aliases, and the *NOALIAS(*p, *q)* indicates that pointers $p$ and $q$ are not aliases. For 166 MAYALIAS and 99 NOALIAS, our technique correctly identified 249 (the accuracy is 94.0%), while Angr-VSA correctly identified 214 cases (the accuracy is 80.8%). In particular, the positive rate of our technique (95.2%) is much higher than that of Angr-VSA (69.9%). Eight MAYALIAS were not correctly identified by our technique for two reasons. First, some fields of complex structure array variables that allocated in the stack cannot be accurately recovered by SSE. Second, our algorithm tracks variables forward and backward on demand, and the lack of contexts from caller function results in missing updates to the store definition. The low accuracy of Angr-VSA is because many variables are unconstrained symbolic values, and Angr-VSA cannot calculate their concrete values. However, *EmTaint* can find the alias relationship of symbolic values with SSE.

In summary, our proposed SSE-based alias analysis has both higher accuracy (94.0%) and true positives (95.2%) in alias analysis compared to VSA (80.8%, 69.9%) in PTABEN dataset.

## 4.3 Indirect Call Resolution with Alias Analysis

In this section, we evaluate the effectiveness and efficiency of indirect call resolution, and also evaluate the effectiveness and necessity of indirect call resolution in discovering taint-style vulnerabilities. Note that, the indirect call resolution procedure is armed with SSE-based alias analysis, Thus, we take SSE-based alias analysis and indirect call resolution as a whole for the evaluation.

**Effectiveness and Efficiency.** Table 6 shows the results of indirect call resolution on our evaluation dataset from Table 3. In Table 6, All I-Calls denotes the total number of unresolved indirect callsite in IDA pro. Resolved I-Calls denotes the the number of indirect callsites that can be resolved by *EmTaint*. The result shows that *EmTaint* can resolve 96.6% indirect callsites (12,022 out of 12,446) and the average analysis time per sample is about 96 seconds, which proves both effectiveness and high efficiency of indirection call resolution module in *EmTaint*. In addition, we calculated the total number of target functions called at indirect callsites during this process as shown in the column I-Call targets. In total, our *EmTaint* 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 failure cases. However, by reverse-engineering some samples, we did find some missed targets because some address-taken functions were not recognized.

**Table 6: Results of indirect call resolution**

| ID | Model | All I-Calls | Resolved I-Calls | I-Call targets | % of resolved I-Calls | Time (s) |
|---|---|---|---|---|---|---|
| 1 | Cisco RV320 | 638 | 620 | 794 | 97.2% | 288.04 |
| 2 | Cisco RV130 | 17 | 14 | 475 | 82.4% | 27.83 |
| 3 | D-Link DIR-825 | 82 | 80 | 239 | 97.5% | 33.44 |
| 4 | D-Link DAP-1860 | 27 | 21 | 327 | 77.8% | 37.15 |
| 10 | TRENDnet TEW632BRP | 43 | 41 | 182 | 95.3% | 20.40 |
| 11 | TRENDnet TEW827RU | 51 | 48 | 381 | 94.1% | 25.56 |
| 12 | NETGEAR R7800 | 17 | 14 | 692 | 82.3% | 40.27 |
| 13 | NETGEAR R8000 | 3 | 2 | 491 | 66.6% | 40.35 |
| 29 | TP-Link WR940NV4 | 389 | 309 | 653 | 79.4% | 409.10 |
| 32 | Tenda AC9V3.0 | 88 | 65 | 286 | 73.9% | 204.55 |
| **Total (35)**§ | - | 12,446 | 12,022 | 14,920 | 96.6% | 95.55† |

§: The row labelled "Total" shows aggregated results for 35 samples. Due to page limit, we only list 10 representative samples in the table. The other 25 samples can be found in Table 6 in [37]. †: The average execution time per sample.

**Effectiveness in Finding Vulnerabilities.** To evaluate the effectiveness of indirect call resolution in finding vulnerabilities, we conduct the comparison with and without indirect call resolution on our dataset. Due to page limit, we show the results of 10 representative samples with real devices in Table 7 in [37]. Here we use four metrics to demonstrate the effectiveness: the number of covered basic blocks, tainted basic blocks, tainted sinks, and generated alerts. We can see that the first three metrics has increased a lot by applying indirect call resolution. That means that indirect call resolution enables *EmTaint* to propagate the tainted data to more functions and variables, and arrive more unsafe sinks. Furthermore, *EmTaint* with indirect call resolution can find 763 more alerts, which correspond to 131 real vulnerabilities. We also study the effectiveness of indirect call resolution on different samples. The results show that *EmTaint* can help generate a number of new alerts in most cases. A special case named TEW827RU does not show improvement on alerts when applying indirect call resolution to it. The reason is that the propagation of tainted data from source to sink does not involve any indirect calls in this sample.

**Necessity in Finding Vulnerabilities.** To evaluate necessity of indirect call resolution, we manually analyzed the 162 real vulnerabilities (11 n-day and 151 0-day) in the 10 samples. Among them, the triggering path of 131 vulnerabilities involves indirect calls, which prove the necessity of applying indirect call resolution to find vulnerabilities in Linux-based embedded firmware.

In summary, *EmTaint* can resolve the indirect call both effectively (96.6% success rate) and efficiently (96s for each sample on average) on dataset. Moreover, we also demonstrate the effectiveness and necessity of indirect call resolution for finding vulnerabilities in Linux-based embedded firmware.

## 4.4 Comparison with KARONTE and SaTC

Three works are closely related to ours, including CodeSonar [20], KARONTE [30] and SaTC [7]. 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 and they perform taint analysis to detect the taint-style vulnerability in embedded firmware. Due to the very similar scope, we reused the dataset [23, 32] released by KARONTE [30] and SaTC [7], which includes 49 firmware samples from 4 embedded vendors (i.e., NETGEAR, D-Link, TP-Link and Tenda). These datasets describe the number of alerts and the number of true positives.

Table 7 shows the results. To be fair, we adopt the same definition of true positive in KARONTE (*cf.* §X.D in [30]). Specifically, 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. It is worth noting that our performance improvement to theirs is in orders of magnitude (3+ hours vs 400+ hours). Moreover, our tool found 22 0-day vulnerabilities in the already extensively tested samples.

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 common taint sources such as `recv` function. Instead, to shorten the path from the sources to sinks, KARONTE infers the taint source by using 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 real sources (e.g., `recv` function), which helps us achieve much higher true positive rates.

In summary, *EmTaint* can produce true alerts with higher accuracy (96%) in less time (less than 4 hour) compared with KARONTE (62%, 451 hours) and SaTC (33%, 459 hours) on the same dataset. Moreover, *EmTaint* can find 22 more 0-day vulnerabilities compared with them.

# 5 DISCUSSION

**Comparison with Existing Work.** First, we compared SSE to existing work that use access path to implement field-sensitive analyses [3, 8, 34]. All these related work are for source code, not binary program. In the source code, all variables and fields have clear symbolic names, making it ideal for representing a memory access through a local variable followed by a sequence of field accesses (e.g., $x.y.z$). However, binary programs lose symbol names, data types, and data structure information of variables, which makes it difficult to use access path directly in binary analysis. For example, for the ARM instructions "`ADD R2, R0, 0x20; ADD R3, R0, 0x48`", register `R2` and `R3` cannot be represented with access-path because it is unknown whether they are pointer or integer. Moreover, it is difficult to identify `R2+0x28` and `R3` as aliases by access-path. To address this issue, we propose SSE. In addition to describing an indirect memory access like access-path, SSE is also used to recovery data types (pointer or non-pointer), calculate offset patterns of base pointer (e.g., identify `load(R2+0x28)` and `load(R3)` as aliases), and abstract recursive data structure and pointer increment structure in the loop (Section 3.2.4). Second, we compared our on-demand alias analyses to existing work [36, 42], which also use demand- or client-driven analyses to improve efficiency. These work compute points-to queries on-demand, which aim to find all objects to which the pointer points. However, our algorithm focuses on finding the

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

| Vendor | Samples | *KARONTE* [30] | | | | *SaTC* [7] | | | | *EmTaint* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Alerts | # of KTP | KTP Rate | Time | Alerts | # of KTP | KTP Rate | Time | Alerts | # of KTP | 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 | 144 | 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".

alias of the pointer itself on-demand, and does not focus on the pointer's target, which is more suitable for taint analysis.

**Limitations.** First, although SSE-based alias analysis outperforms existing alias techniques in binary level, it still falls short when the pointer involves bitwise operations or the offset of the memory access is not a constant. We will improve SSE to handle this situation in the future work. Second, for vulnerability detection, *EmTaint* generates 14% false positives. The reason is manifold. (1) *EmTaint* misses some constraint checks that exist in customized library functions. Specifically, our approach collects constraint checks in the target program itself and standard library function, but misses security checks that do occur in library functions without summaries. This problem can be mitigated by manual providing the function summaries for customized library functions. (2) *EmTaint* incorrectly recovers some indirect calls that do not exist, which leads to infeasible paths. This problem can be solve by applying dynamic analysis to filter the results (e.g., dynamic symbolic execution can be used to solve the path constraints). We leave this as our future work. (3) *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.

## 6 RELATED WORK

**Alias Analysis.** Alias analysis is a long-term research topic in source code analysis [1, 8, 16, 35, 36, 42]. However, alias analysis in binary is not as advanced as source code analysis. Debray et al. [15] proposed an inter-procedural flow-sensitive pointer alias analysis for *x*86 executables, which is context-**in**sensitive. Guo et al. [21] presented the first context-sensitive points-to analysis for *x*86 assembly code, which is only partially flow-sensitive. Reps et al. [4, 31] utilized value-set analysis (VSA) to identify pointer alias through tracking memory accesses in *x*86 executables. However, VSA is expensive and unpractical for real-world complex programs [43]. To adapt to complex binaries, BDA [43] proposed to utilize path sampling to generate accurate data dependency. However, it does not cover all path and is limited by path depth, which makes it unable to guarantee the robustness of pointer analysis. BinPointer [24] utilizes an offset-sensitive block memory model to implement inter-procedural pointer analysis, which is flow-**in**sensitive on memory locations and context-**in**sensitive. We proposed a new alias analysis technique based on SSE. To the best of our knowledge, this is the first work that simultaneously achieves on-demand, flow-, context- and field-sensitive alias analysis for binary.

**Vulnerability Detection Techniques for Embedded System.** For vulnerability detection of embedded system, there are both static approaches [7, 9, 12, 14, 19, 25, 30, 33, 40] and dynamic approaches [6, 26, 27, 38, 44–46]. In static approaches, since our research scope focuses on using taint analysis techniques to detect vulnerabilities in Linux-based firmware without source code, only DTaint [9], KARONTE [30] and SaTC [7] can be applied to our scenario. DTaint 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, DTaint lacks accuracy and efficiency in data flow analysis. KARONTE 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 also performs taint analysis to discover vulnerabilities in embedded systems. It utilizes shared keywords related to user input in the front-end and back-end to infer the taint source. All the aforementioned works that detect taint-style vulnerabilities did not perform indirect call resolution, which we have demonstrated to be critical in §4.3.

## 7 CONCLUSION

In this work, we propose *EmTaint*, a novel static approach for accurate and fast detection of taint-style vulnerabilities in embedded firmware. The key techniques in *EmTaint* is SSE-based on-demand alias analysis, which facilitates indirect call resolution and accurate taint analysis. We implemented the prototype of *EmTaint* and evaluated it against 35 real-world embedded firmware samples from six popular vendors. The evaluation result shows that *EmTaint* discovered at least 192 vulnerabilities, including 41 n-day vulnerabilities and 151 0-day vulnerabilities. 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 vulnerabilities on the same dataset with high accuracy in less time.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments to improve our paper. This work was partially supported by National Key R&D Program of China (No. 2022YFB3103904), NSF CNS-2019340, NSF ECCS-2140175, a grant from Cisco Research, National Natural Science Foundation of China (No. 62072451, 92267105), Guangdong Special Support Plan (No. 2021TQ06X990), and Shenzhen Basic Research Program (No. JCYJ20200109115418592 and JCYJ20220818101610023).

## DATA-AVAILABILITY STATEMENT

All tools and detailed instructions to reproduce our study are openly available from Zenodo [17].

## REFERENCES


[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer.

[2] Angr 2023. *Next-generation binary analysis framework.* Retrieved May 22, 2023 from https://github.com/angr/angr

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (jun 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[4] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You EXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (aug 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[5] Bertrand Meyer 2019. *Soundness and Completeness: Defined With Precision.* Retrieved May 22, 2023 from https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-defined-with-precision/fulltext

[6] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS.* http://dx.doi.org/10.14722/ndss.2018.23159

[7] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 303–319. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-libo

[8] Ben-Chung Cheng and Wen-Mei W. Hwu. 2000. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. *SIGPLAN Not.* 35, 5 (may 2000), 57–69. https://doi.org/10.1145/358438.349311

[9] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018.* IEEE Computer Society, 430–441. https://doi.org/10.1109/DSN.2018.00052

[10] Claripy 2023. *The API documentation provided by Claripy.* Retrieved May 22, 2023 from https://docs.angr.io/projects/claripy/en/latest/api.html

[11] CLE 2023. *A python module for loading binaries.* Retrieved May 22, 2023 from https://github.com/angr/cle

[12] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 309–326. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[13] CVE 2023. *Common vulnerabilities and exposures.* Retrieved May 22, 2023 from https://cve.mitre.org/

[14] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, Samuel T. King (Ed.). USENIX Association, 463–478. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson

[15] Saumya K. Debray, Robert Muth, and Matthew Weippert. 1998. Alias Analysis of Executable Code. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 12–24. https://doi.org/10.1145/268946.268948

[16] Alain Deutsch. 1994. Interprocedural May-Alias Analysis for Pointers: Beyond $k$-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 230–241. https://doi.org/10.1145/178243.178263

[17] EmTaint 2023. *Reproduction Package for Article 'Detecting Vulnerabilities in Linux-based Embedded Firmware with SSE-based On-demand Alias Analysis'.* Retrieved May 27, 2023 from https://doi.org/10.5281/zenodo.7976968

[18] EXPLOIT DATABASE 2023. *Exploit database of the website.* Retrieved May 22, 2023 from https://www.exploit-db.com/

[19] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 896–899. https://doi.org/10.1145/3238147.3240480

[20] Grammatech 2023. *A source code and binary code static analysis tool.* Retrieved May 22, 2023 from https://www.grammatech.com/our-products/codesonar/

[21] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. 2005. Practical and Accurate Low-Level Pointer Analysis. In *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA.* IEEE Computer Society, 291–302. https://doi.org/10.1109/CGO.2005.27

[22] IDA Pro 2023. *A powerful disassembler.* Retrieved May 22, 2023 from https://www.hex-rays.com/ida-pro/

[23] Karonte 2020. *The experimental dataset used by tool Karonte.* Retrieved May 22, 2023 from https://github.com/ucsb-seclab/karonte#dataset

[24] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 169–180. https://doi.org/10.1145/3497776.3517776

[25] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. $\alpha$Diff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 667–678. https://doi.org/10.1145/3238147.3238199

[26] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, Vol. 18. 1–11.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society. http://dx.doi.org/10.14722/ndss.2018.23166

[28] PTABEN 2022. *A micro-benchmark suite designed for validating various static analysis algorithms.* Retrieved May 22, 2023 from https://github.com/SVF-tools/Test-Suite

[29] PyVEX 2023. *A python module for VEX intermediate represenation.* Retrieved May 22, 2023 from https://github.com/angr/pyvex

[30] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 1544–1561. https://doi.org/10.1109/SP40000.2020.00036

[31] Thomas W. Reps and Gogul Balakrishnan. 2008. Improved Memory-Access Analysis for x86 Executables. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4959)*, Laurie J. Hendren (Ed.). Springer, 16–35. https://doi.org/10.1007/978-3-540-78791-4_2

[32] SaTC Dataset 2022. *The experimental dataset used by tool SaTC.* Retrieved May 22, 2023 from https://drive.google.com/file/d/1rOhjBlmv3jYmkKhTBJcqJ-G56HoHBpVX/view

[33] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.* The Internet Society. http://dx.doi.org/10.14722/ndss.2015.23294

[34] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL (2019), 48:1–48:29. https://doi.org/10.1145/3290361

[35] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 32–41. https://doi.org/10.1145/237721.237727

[36] Yulei Sui and Jingling Xue. 2020. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++. *IEEE Trans. Software Eng.* 46, 8 (2020), 812–835. https://doi.org/10.1109/TSE.2018.2869336

[37] Supplementary material 2023. *A supplementary material for paper 'Detecting Vulnerabilities in Linux-based Embedded Firmware with SSE-based On-demand Alias Analysis'.* Retrieved May 25, 2023 from https://drive.google.com/file/d/15K3Nbqm15sTwfXvMuiwhwBqvFTscZ1cy/view

[38] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. 2013. RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing. *KSII Trans. Internet Inf. Syst.* 7, 8 (2013), 1989–2009. https://doi.org/10.3837/tiis.2013.08.014

[39] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 6913)*, Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis (Eds.). Springer, 522–536. https://doi.org/10.1007/978-3-642-23808-6_34

[40] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer*

*and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 363–376. https://doi.org/10.1145/3133956.3134018

[41] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 797–812. https://doi.org/10.1109/SP.2015.54

[42] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 155–165. https://doi.org/10.1145/2001420.2001440

[43] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 137:1–137:31. https://doi.org/10.1145/3360563

[44] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[45] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 2022. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 417–428. https://doi.org/10.1145/3533767.3534414

[46] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. 2019. An Efficient Greybox Fuzzing Scheme for Linux-based IoT Programs Through Binary Static Analysis. In *38th IEEE International Performance Computing and Communications Conference, IPCCC 2019, London, United Kingdom, October 29-31, 2019*. IEEE, 1–8. https://doi.org/10.1109/IPCCC47392.2019.8958740