Security Analysis of IoT Frameworks using Static Taint Analysis

Tuba Yavuz
ECE Department
University of Florida
USA
tuba@ece.ufl.edu

Christopher Brant
ECE Department
University of Florida
USA
cdbrant@ufl.edu

ABSTRACT

Internet of Things (IoT) frameworks are designed to facilitate provisioning and secure operation of IoT devices. A typical IoT framework consists of various software layers and components including third-party libraries, communication protocol stacks, the Hardware Abstraction Layer (HAL), the kernel, and the apps. IoT frameworks have implicit data flows in addition to explicit data flows due to their event-driven nature. In this paper, we present a static taint tracking framework, IFLOW, that facilitates the security analysis of system code by enabling specification of data-flow queries that can refer to a variety of software entities. We have formulated various security relevant data-flow queries and solved them using IFLOW to analyze the security of several popular IoT frameworks: Amazon FreeRTOS SDK, SmartThings SDK, and Google IoT SDK. Our results show that IFLOW can both detect real bugs and localize security analysis to the relevant components of IoT frameworks.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Software security engineering**; • **Software and its engineering → Software verification and validation**.

KEYWORDS

taint tracking, IoT, static analysis

ACM Reference Format:
Tuba Yavuz and Christopher Brant. 2022. Security Analysis of IoT Frameworks using Static Taint Analysis. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY '22), April 24–27, 2022, Baltimore, MD, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3508398.3511511

1 INTRODUCTION

Internet of Things (IoT) frameworks are designed to facilitate provisioning of IoT devices, which includes the management of access to data and resources by these devices and their secure configuration. Software vulnerabilities in IoT frameworks form an important part of the IoT attack surface. Attackers can exploit such vulnerabilities to infect IoT devices with DDos botnets [5], leak sensitive data [2], or destroy IoT devices [8].

CODASPY '22, April 24–27, 2022, Baltimore, MD, USA

ACM ISBN 978-1-4503-9220-4/22/04…$15.00
https://doi.org/10.1145/3508398.3511511

An IoT framework typically consists of third-party code such as cryptographic libraries, e.g., mbedTLS , messaging protocol libraries, e.g. MQTT , data interchange libraries, e.g., JSON , etc. In addition to the vulnerabilities within the third-party libraries, an IoT framework may host vulnerabilities due to misuse of third-party API or internal APIs, memory vulnerabilities, and leakage of sensitive data.

In this paper, we present a static taint analysis approach that improves existing static taint analysis approaches [9, 11, 19, 22] by supporting a richer set of software entities in the formulation of data-flow queries and by supporting two types of sanitization mechanisms.

Unlike previous works that perform data-flow analyses on IoT apps [10, 11, 26], we focus on the IoT framework Software Developement Kits (SDKs). Although firmware rehosting and firmware fuzzing [12, 14, 27] can help find vulnerabilities, these approaches analyze the attack surface mainly with respect to peripheral interaction. However, an important part of the attack surface is due to the use of APIs and the flow of sensitive data within the framework.

We have implemented our static taint tracking approach in a tool called, IFLOW, using the SVF tool [22]. Our taint tracking supports fine granular source/sink entities in the form of specific data fields and specific function arguments. IFLOW allows specification of sanitization in the form of functions with specific return values or in the form of branching instructions. IFLOW can formulate both API misuse vulnerabilities and sensitive data flows.

We have applied IFLOW to three popular IoT Frameworks: Amazon FreeRTOS SDK [3], SmartThings SDK [7], and Google IoT [6]. We have formulated over 150 API Misuse queries and over 100 sensitive leak queries. Our analysis found an API misuse vulnerability in mbedTLS and potential leakage of various credentials such as WiFi passwords. IFLOW can both detect privacy issues and help better understand the API.

We have equipped IFLOW with several optimizations to scale our analysis to our case studies. IFLOW computes function summaries based on the Sparse Value Flow Graph generated by SVF and performs Contex-Free Language reachability while leveraging these summaries. IFLOW also supports other configurations to optimize the query solving through bounded search and reduced search.

This paper makes the following contributions:

- Various data-flow query types with sanitization options that are expressive for formulating API misuse vulnerabilities and leakage of sensitive data types.
- An open-source[1] static data-flow query engine, IFLOW, for C programs developed on top of the SVF tool.

[1] IFLOW is available at https://github.com/sysrel/IFLOW.

```
1  // Establish the connection
2  if (xStatus == pdPASS)
3     if (SOCKETS_Connect(pxConnection->xSocket, ...))
4          != SOCKETS_ERROR_NONE)
5        xStatus = pdFAIL;
6  if (xStatus == pdPASS) { ... }
7  else prvGracefulSocketClose(pxConnection);
8  ////////////////////////////////////
9  // Called from SOCKETS_Connect
10 while (0 != (xResult = mbedtls_ssl_handshake(...))) {
11   if ( (MBEDTLS_ERR_SSL_WANT_READ != xResult) &&
12        (MBEDTLS_ERR_SSL_WANT_WRITE != xResult))
13         break;
14   ...
15 }
16 ////////////////////////////////////
17 // Called from prvGracefulSocketClose
18 while (xRead < xReadLength) {
19    xResult = mbedtls_ssl_read(...)
20 }
```

Figure 1: The vulnerable (CVE-2018-16528) code in Amazon FreeRTOS that misuses mbedTLS API.

- Security analysis of three IoT framework SDKs and a large set of API misuse and sensitive leakage queries that can be used for new IoT applications that will be deployed on these frameworks or for the evolution of these frameworks.

This paper is organized as follows. We present a motivating API misuse vulnerability in Section 2. We present the technical details of our approach in Section 3 by presenting the supported query types and the algorithms that solve these queries. In Section 4, we present our findings on applying IFLOW over 300 queries across the three IoT frameworks. We discuss the limitations of our approach in Section 5. In Section 6, we position our work in the context of related work. In Section 7, we conclude with directions for future work.

2 MOTIVATION

In this Section, we present an API misuse vulnerability [1] that led to a double-free (CVE-2018-16528) within two modules of Amazon FreeRTOS: MQTT Agent and Green Grass Discovery. The misuse involves two mbedTLS functions: `mbedtls_ssl_handshake` and `mbedtls_ssl_read`. Figure 1 shows the code snippet that misuses the API. The problem arises when `SOCKETS_Connect` (line 3) returns an error value due to a failed `mbedtls_ssl_handshake` (lines 10-13). The error case is handled by the `prvGracefulSocketClose` function, which calls the `mbedtls_ssl_read` function on a corrupted SSL context. However, according to the documentation provided in mbedTLS and partially shown in Figure 2, when `mbedtls_ssl_handshake` returns anything other than 0 or the four specific cases shown on lines 4-7 in Figure 2, then SSL context should not be used, e.g., `mbedtls_ssl_read` and `mbedtls_ssl_write` should not be called on the same context unless `mbedtls_ssl_session_reset` has been called on it.

IFLOW can detect this type of API misuse in a precise way by reasoning about the return value related conditions and by reporting only those that violate the API rules.

3 APPROACH

In this section, we present the technical details of our taint tracking approach. We introduce some background about static data-flow analysis as implemented in SVF [23] in Section 3.1. In Section 3.2, we introduce the type of queries IFLOW supports and in Section 3.3, we explain how IFLOW implements these queries using SVF. In Section 3.4, we explain some optimizations we provided in IFLOW to improve its scalability.

```
1  * warning
2  *       If this function returns something other than
3  *      0,
4  *      MBEDTLS_ERR_SSL_WANT_READ,
5  *      MBEDTLS_ERR_SSL_WANT_WRITE,
6  *      MBEDTLS_ERR_SSL_ASYNC_IN_PROGRESS or
7  *      MBEDTLS_ERR_SSL_CRYPTO_IN_PROGRESS,
8  *      you must stop using the SSL context for reading or writing,
9  *      and either free it or call mbedtls_ssl_session_reset()
10 *      on it before re-using it for a new connection; the current
11 *      connection must be closed.
```

Figure 2: Documentation about `mbedtls_ssl_handshake` in mbedTLS.

3.1 Static Data-flow Analysis

Static data-flow analysis typically uses a graph based representation of a program to statically determine the values that may flow into a variable at a certain program location. One such program representation is the System Dependence Graph (SDG) [17], where vertices represent program statements and the edges represent control-flow or data-flow dependence between program statements.

In this work, we use SVF to generate a Sparse Value Flow Graph (SVFG), which is an SDG without the control-flow edges[2]. SVF uses the LLVM IR and the results of its points-to analysis to build the SVFG. Specifically, it builds a Static Single Assignment (SSA) form, called Memory SSA, for address-taken variables that represent abstract memory objects. Memory SSA form reveals the def-use dependencies for address taken variables and complements the SSA representation for LLVM registers, whose def-use chains are already computed within the LLVM IR.

An SVFG has two types of edges: direct and indirect. The former refers to def-use dependencies that are available in the LLVM IR and the latter refers to def-use dependencies computed based on the Memory SSA. SVF provides additional information about the nodes such as the type of program statement and about the edges such as whether the dependency is intra-procedural or inter-procedural (through a parameter or a return value at a callsite).

```
1  char *receive(char *buf) {
2    buf[0] = 'A';
3    return buf;
4  }
5  int main(int argc, char **argv) {
6      char *buf1 = (char*)malloc(10);
7      char *buf2 = (char*)malloc(10);
8      char *buf3 = (char*)malloc(10);
9      char *b = receive(buf1);
10     int length = buf1[0];
11     memcpy(buf2, b, buf1 - buf2);
12     memcpy(buf3, b, length);
13     ...
14 }
```

Figure 3: Sample code with data-flows through pointers.

[2]SVF provides the inter-procedural control-flow graph (ICFG) and the control-flow graph (CFG) for each function can be constructed through the LLVM interface.

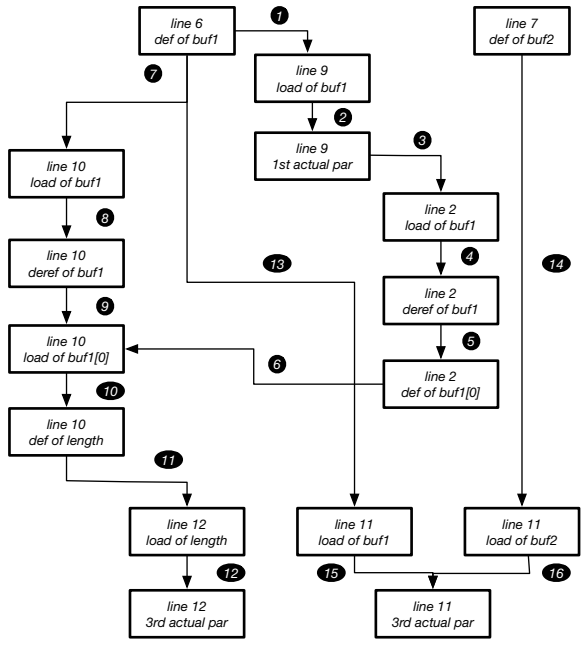

Figure 4: Part of the sparse value flow graph for the example in Figure 3.

Figure 3 shows a sample code that has data-flow through a pointer, `buf1`, as well as through a function call, `receive`. Figure 4 shows part of the sparse value flow graph and in simplified form. SVF recognizes memory allocation sites such as lines 6 and 7 as the definition of abstract memory objects and keeps track of data flows of these memory objects. It does so both intraprocedurally and interprocedurally. For instance, since the `receive` function sets the first element of its formal parameter, `buf`, and `buf1` is passed as an actual parameter to `receive` at line 9, the dependency between line 2 and line 10 will be captured through the path 1-2-3-4-5-6-10-11-12 in Figure 4. SVF represents dereferencing operations, i.e., a sequence of `load` instruction followed by a `GetElementPtr` instruction or another load instruction in LLVM, explicitly in the SVFG. So, the existence of a path from a source node to a destination node may denote direct flows as well as indirect flows through one or more levels of indirection through the dereferencing operations. For instance, the path 6-10 in Figure 4 represents the direct data-flow between line 2 and line 10 in Figure 3 where as the path 7-8-9-10 in Figure 4 represents the indirect data-flow between line 6 and line 10 in Figure 3.

3.2 Query Types

Since our goal is to support security analysis of frameworks, we present several data-flow query types that can be used to model various vulnerability types. Our goal is to minimize the number of false positives as well as false negatives. The former requires precise formulation of the dependencies. The latter requires considering the possibility of the arbitrary data-flows due to the interaction of arbitrary data-structures. In what follows, we explain the query types we consider in this work through diagrams that depict a combination of data-flow and control-flow dependencies. We use dashed arrows to denote control-flow dependence, solid arrows to denote direct data-flow dependence, i.e., no dereferencing operation between the source and the sink, and a dotted arrow to denote a possibly indirect data-flow.

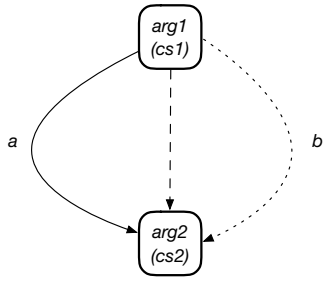

Figure 5: Query Type 1: Source a) directly (solid arrow) or b) indirectly (dotted arrow) taints the sink. Dashed arrow denotes control-flow.

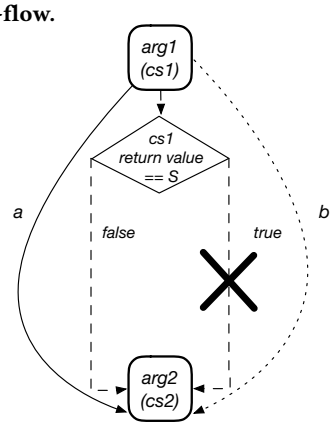

Figure 6: Query Type 2: Source a) directly (solid arrow) or b) indirectly (dotted arrow) taints the sink while subject to sanitization rule. arg1 (arg2) is an actual argument of callsite cs1 (cs2). S denotes the sanitization value. Dashed arrow denotes control-flow.

Figure 5 depicts the first type of query IFLOW supports. The source and the sink can be any argument of a function at some callsite or a specific argument specified via the position. IFLOW also enables specifying the type of the argument as one of the following types: primitive, data pointer, function pointer, or struct type. The source may also be specified as a specific field of a data type. This type of query has two subtypes: Type 1.a and Type 1.b denote direct and indirect data flows, respectively.

Figure 6 depicts the second type of query, which extends Type 1 query with a sanitization rule. When the source is an argument of a function and the data-flow query rules out cases when the function returns a specific value, which we call the sanitization value and denote with S, Type 2 queries should be used. As shown by the cross symbol in Figure 6, paths on which the function called at callsite $cs1$ returns S are not returned included in the query solution set.

As an example, in Figure 7 there is a direct data flow from the first argument of `foo1` callsite at line 9 (`ctx1`), and the first argument of `bar1` callsites at lines 10 and 12, (`d1->sc`) due to the assignment at line 2. So, a Type 1.a query would yield the paths 9-10 and 9-12. However, if we specify a Type 2.a query, where we use 0 as the

```
1  int foo1(struct context *ctx, struct data *res) {
2    res->sc = ctx;
3    if (cond1)
4       return 0;
5    else return 1;
6  }
7  void bar1(struct context *ctx,...) {...}
8  int main(...) {
9    if (foo1(ctx1, d1))
10       bar1(d1->sc, v1);
11   else
12       bar1(d1->sc, v2);
13 }
```

Figure 7: Sample code with data flows between the source and the sink.

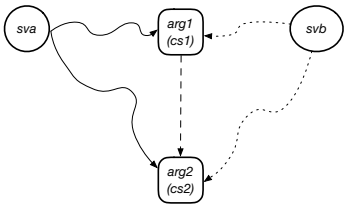

Figure 8: Query Type 3: Source and sink are a) directly (solid arrow) or b) indirectly (dotted arrow) tainted by some common value. arg1 (arg2) is an actual argument of callsite cs1 (cs2). The dashed arrow denotes control-flow.

sanitization value then the solution includes only the path 9-10 as the path 9-12 would be eliminated due to the sanitization rule.

Figure 8 shows the third type of query, where the data is not required to flow between the entities of interest, e.g., `arg1` and `arg2`, and, instead, the data flows from a common value into these entities. As an example, in Figure 9, `conf− >sc` flows both into the first argument of `foo2` callsite at line 10 and to the first arguments of `bar2` callsites at lines 11 and 13. However, there is no data flow from the first argument of `foo2` callsite at line 10 to the first arguments of `bar2` callsites at lines 11 and 13. Therefore, a Type 1 query that specifies `foo2`'s first argument as the source and `bar2`'s first argument as the sink would have an empty solution. However, if we intend to locate paths like 10-11 or 10-13, where `foo2` and `bar2` receive the same value, which corresponds to `conf->sc` in Figure 9 and is denoted by `sva` or `svb` in Figure 8, then we should formulate a Type 3 query.

```
1  int foo2(struct context *ctx) {
2    if (ctx->a)
3       return 0;
4    else return 1;
5  }
6  void bar2(struct context *ctx) {...}
7  int main(...) {
8    struct data *conf = ...;
9    ctx1 = conf->sc;
10   if (foo2(ctx1))
11       bar2(conf->sc);
12   else
13       bar2(conf->sc);
14 }
```

Figure 9: Sample code with data flows where the source and the sink are tainted by a common value.

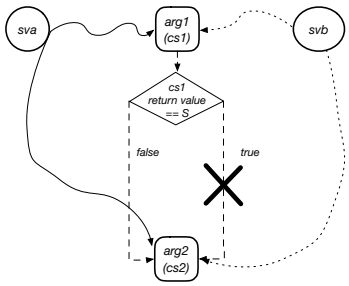

Figure 10: Query Type 4: Source and sink are a) directly (solid arrow) or b) indirectly (dotted arrow) tainted by some common value while subject to sanitization rule. arg1 (arg2) is an actual argument of callsite cs1 (cs2). S denotes the sanitization value. The dashed arrow denotes control-flow.

Figure 10 depicts the fourth type of query, which extends Type 3 queries with a sanitization rule. So, for the code in Figure 9, a Type 4 query where `arg1` and `arg2` are specified as the first arguments of `foo2` and `bar2` callsites, respectively, and the sanitization rule of 1 is specified then the path 10-13 would be returned in the solution and not the path 10-11.

3.3 Query Engine

We have implemented a query engine to solve the four types of data-flow queries presented in Section 3.2. We configured SVF to use context-insensitive points-to analysis to scale it to large code bases. We used the Sparse Value Flow Graph (SVFG) and the Inter-Procedural Control-Flow Graph (ICFG) generated by SVF to solve the queries. We use Context-Free Language (CFL) Reachability algorithm [20] for checking reachability both in the SVFG and in the ICFG to consider only valid paths, i.e., updates the calling context at function entry nodes and follows only the paths that match the calling context at function exit node.

Algorithm 1 shows how the query types 1 and 2 are solved. Given a query $Q = (Arg_1, Arg_2, San, S)$, where Arg_1 (Arg_2) is the set of arg_1 (arg_2) nodes as shown in Figures 5 and 6, for each pair (arg_1,arg_2) it checks if there is a direct or indirect data-flow between them. If so, it checks for the existence of the control-flow. Note that if there is a data-flow there must be a control-flow path between the two entities. However, when sanitization is involved, control-flow check ensures that there is still a valid path after eliminating paths due to sanitization and whether the pair should be included in the solution set.

Algorithm 2 solves the query types 3 and 4. Given a query $Q = (Arg_1, Arg_2, San, S)$, where Arg_1 (Arg_2) is the set of arg_1 (arg_2) nodes as shown in Figures 8 and 10, for each pair (arg_1,arg_2) it checks if there is some value in the SVFG from which data-flows into both arg_1 and arg_2. Once such common value is found, existence of a control-flow path between arg_1 and arg_2 is check subject to sanitization rules.

In both Algorithm 1 and 2, Algorithm 3 is used to compute the set of basic blocks, *Sanbb*, that is used to disqualify the valid yet uninteresting control-flow paths between arg_1 and arg_2. For query types 2 and 4, *San* denotes the set of callsites that are used as sanitizers when the return value is equal to S. For each callsite cs in *San*, we check whether it is a control-flow relevant instruction,

Algorithm 1: Solving Query Types 1 and 2.

Input: Q: Query, G_1: SVFG, G_2; ICFG, $Type$: {Direct, Indirect}
Output: Sol: Set of Satisfying Node Pairs

$Sol \leftarrow \emptyset$;
Let $Q = (Arg_1, Arg2_2, San, S)$ ▷ $San = \emptyset$ and $S = undef$ for Type 1
$Sanbb \leftarrow FindSan(San, S, ICFG)$
for *each* $arg_1 \in Arg_1$ **do**
  for *each* $arg_2 \in Arg_2$ **do**
    if *exists a path p from* arg_1 *to* arg_2 *in* G_1 **then**
      if $Type$ *is Direct and there exists a dereferencing node on p* **then**
        continue
      end
    end
    if *there exists a path from* $BB(arg_1)$ *to* $BB(arg_2)$ *in* G_2 *without going through any of the nodes in Sanbb* **then**
      $Sol \leftarrow Sol \cup \{(arg_1, arg_2)\}$
    end
  end
end

Algorithm 2: Solving Query Types 3 and 4.

Input: Q: Query, G_1: SVFG, G_2; ICFG, $Type$: {Direct, Indirect}
Output: Sol: Set of Satisfying Node Pairs

Let $Q = (Arg_1, Arg2_2, San, S)$ ▷ $San = \emptyset$ and $undef$ for Type 3
$Sanbb \leftarrow FindSan(San, S)$
for *each* $arg_1 \in Arg_1$ **do**
  for *each* $arg_2 \in Arg_2$ **do**
    if $Type$ *is Direct* **then**
      Let P_1 (P_2) denote the set of nodes from which there is a path to arg_1 (arg_2) in G_1 without visiting a dereferencing node
    end
    else
      Let P_1 (P_2) denote the set of nodes from which there is a path to arg_1 (arg_2) in G_1
    end
    if $P_1 \cap P_2 \neq \emptyset$ **then**
      if *there exists a path from* $BB(arg_1)$ *to* $BB(arg_2)$ *in* G_2 *without going through any of the nodes in Sanbb* **then**
        $Sol \leftarrow Sol \cup \{(arg_1, arg_2)\}$
      end
    end
  end
end

Algorithm 3: FindSan: Finding sanitization nodes.

Input: San: Sanitization CallSites, S: Sanitization Value
Output: $Sanbb$: Sanitization Basic Blocks

$Sanbb \leftarrow \emptyset$
for *each* $cs \in San$ **do**
  if *cs do not return a value or is not control-flow relevant* **then**
    continue
  end
  if *return value of cs, r, flows into a conditional branch inst. br* **then**
    Let bb denote the basic block terminated by br
    if $r = S$ *condition flows into br* **then**
      $Sanbb \leftarrow Sanbb \cup \{TrueTarget(bb)\}$
    end
    else if $r \neq S$ *condition flows into br* **then**
      $Sanbb \leftarrow Sanbb \cup \{FalseTarget(bb)\}$
    end
    else
      skip ▷ Could not be decided
    end
  end
  else
    ▷ Return value of cs flows into a return inst. ret
    Let cs' denote the callsite that ret returns to
    return $Sanbb \leftarrow Sanbb \cup FindSan(\{cs'\}, S, G_1, G_2)$
  end
end

i.e, returns a value and the value flows into some branch condition. We leverage the use-def chains to find out where the return value of *cs*, if any, flows. We check the LLVM comparison instruction `ICmpInst` and the operators *ICMP_EQ* and *ICMP_NE* for equality and inequality checking predicates. We precisely compute the result of the comparison when the value is compared against a constant value. If the return value of *cs* is equal to *S* then we include the target basic block of the true branch, and that of the false branch otherwise. In case the comparison is more complicated then we conservatively do not record any sanitization basic blocks, which may lead to false positives in the query solutions. Although Algorithm 3 works for a single sanitization value, IFLOW supports specifying multiple sanitization values, which would be needed, for instance, for the mbedTLS API misuse presented in Section 2. In such a case, for each sanitization value the set of basic blocks are computed and a union of these basic blocks are used while filtering out the valid source-sink patterns.

3.4 Query Optimizations

Solving data-flow queries for large frameworks is challenging as scalability becomes an issue. We have implemented several optimization strategies to reduce query solving time.

Function Summaries. Before solving the query, we traverse the SVFG and detect data-flows from the formal parameters to the outside contexts. We keep a map from the formal parameter nodes to the boundary nodes, `FormalOut` and `FormalRet` type nodes, that establish data flows to outside the function. By keeping maps from `FormalIn` type of nodes to a set of `FormalOut` or `FormalRet` type nodes, we skip traversing intermediate nodes when checking for data-flow reachability. However, we make sure to traverse the internal nodes for a function when the function includes arg_1 (the source) or arg_2 (the sink) to avoid missing a path due to summarization. One of the maps stores reachability information for direct data flows and another stores the same for indirect data flows.

Bounded Search. We set a bound for data-flow reachability as well as control-flow reachability. So, the search terminates when the bound is reached leading to under approximation.

4 EVALUATION

We have chosen three open-source IoT framework SDKs: Amazon FreeRTOS SDK, SmartThings SDK, and Google IoT SDK. These SDKs are written to facilitate development of IoT device applications. Our goal is to analyze the attack surface of these SDKs using data-flow analysis.

Table 1 shows various characteristics of our benchmarks. We compiled these projects with the clang compiler, used the default configurations[3], and generated the LLVM bitcode as SVF, on which IFLOW is built on, works at the LLVM IR level. We have computed the source lines (SLOC) using David Wheeler's sloc tool. For Amazon FreeRTOS, we have included the libraries directory and excluded the projects directory as the latter mainly includes vendor specific code. However, we did compile Amazon FreeRTOS for the STM 32l475 board and, therefore, included STM 32l475 specific code as well. We compiled SmartThings SDK and Google IoT SDK without choosing a particular target due to challenges of compiling the platform specific code for the available ports. We present the size of the LLVM bitcode since only a part of the source gets compiled due to the configuration settings. We list the number of source files that get transformed into LLVM bit code under the LLVM Modules column. We measured the sizes of the Call Graph and the Sparse Value Flow Graph in terms of the number of nodes and edges.

We have run our experiments on an Ubuntu machine with 32 GB RAM with Intel i7 Core. We set the reachability bound to 50 as the default value.

We evaluated IFLOW in terms of its effectiveness in answering security queries and its performance.

4.1 Security Queries

4.1.1 Leakage of Sensitive Data. We want to understand what type of security sensitive information gets stored, generated, and processed by these frameworks and whether these sensitive information would be leaked. We analyzed the header files for the data types and identified sensitive fields. Table 2 lists the `struct` data types and the sensitive fields along with the type of information stored in these fields. We have encountered private/secret keys, WiFi passwords, Pin codes, and MQTT client passwords.

[3]With the exception of SmartThings SDK for which we were able to enable the DEBUG options.

We have formulated a data-flow query using Query Type 1 in a way that the sensitive field accesses were designated as the source nodes and and the arguments of `memcpy` and functions that include one of the keywords `Write`, `Print`, `Dump`, `Log`, `Publish`, and `Send` (along with their all lowercase and all uppercase versions) as the sinks. Although we did not expect any explicit leakage of sensitive data within the framework code, we wanted to find out how sensitive data moved around, e.g., via `memcpy`, and in which contexts they get accessed. We present our findings below.

4.1.2 Google IoT SDK. Google IoT implements an event-based architecture for I/O. It copies the sensitive fields within the MQTT layer using memcpy and places the generated connection data to some buffer, which gets added to an I/O event queue to be processed later. This is probably why we did not find direct data flows to other sinks. In the contexts that data copying was performed, e.g., `do_mqtt_connect`, we found out that the buffers with sensitive data (connection password, a field of the `iotc_connection_data_s` type) were freed without clearing the contents. Although as a security precaution the `IOTC_SAFE_FREE` macro sets the pointer to `NULL` after freeing the memory, the free operation is implemented in terms of the platform specific free function. In none of the platform specific implementations the memory gets cleared.

4.1.3 SmartThings SDK. Inside the `iot_nv_get_wifi_prov_data` function, the password is read from NVRAM and then some other elements of the provisioning info such as MAC string and BSSID are read. If there are any conversion errors encountered with these, the password is not erased from the data structure that was previously initialized with the password read from the NVRAM. What makes this complicated is that such conversion errors are not propagated to the calling contexts of this function as IOT_ERROR_NONE is returned.

Inside the `_iot_security_be_software_pk_sign_ed25519` function, the secret key is copied to a local array, which does not get cleared upon function return.

When CONFIG_STDK_IOT_CORE_LOG_LEVEL_DEBUG is enabled, then sensitive information such as the Wifi password would get written to the standard output. While some project examples for the SmartThings SDK provide two separate configurations: a debug configuration and a non-debug one. In the debug configurations this option is enabled, However, in the rtl8721c project example, a single configuration with the CONFIG_STDK_IOT_CORE_LOG_LEVEL_DEBUG option being enabled is provided. Although Realtek provides a disclaimer about their project in their website [4], we think that it is safer to separate the debug configurations from non-debug ones to prevent accidental enabling of such options during deployment. Also, SmartThings SDK leaks the address of some sensitive data in DEBUG mode.

4.1.4 Amazon FreeRTOS. Inside the STM 32l475 specific implementation of the `WIFI_ConnectAP` function, Wi-Fi password is copied to a local array to be used as a parameter for the `ES_WIFI_IsConnected` function. However, this local array that stores the password is not cleared upon function return. IFLOW also reports a data-flow within the STM 32l475 specific implementation of the `WIFI_ConfigureAP` function, which has the same problem. Since we were only able to compile Amazon FreeRTOS for the STM 32l475 port,

Framework	SLOC	LLVM Bitcode	LLVM Modules	Call Graph		SVFG	
				#Nodes	#Edges	#Nodes	#Edges
Amazon FreeRTOS SDK	452K	2075KB	266	3220	12748	363108	540581
SmartThings SDK	193K	4066KB	241	2613	12994	287124	433924
Google IoT SDK	199K	783KB	51	429	2396	44709	58825

Table 1: Benchmark characteristics.

Framework	Data Type	Field	Type of Sensitive Data
Amazon FreeRTOS	IotNetworkCredentials	pPrivateKey	Private Key
	P11KeyConfig_t	uxDevicePrivateKey	Private Key
	IotHttpsConnectionInfo	pPrivateKey	Private Key
	IotHttpsAsyncInfo	pPrivData	Private Data
	BTPinCode_t	ucPin	Bluetooth PinKey Code
	WIFINetworkParams_t	xPassword	Wi-Fi WEP Key WPA/WPA2 Pass Phrase
	WIFINetworkProfile_t	cPassword	Wi-Fi Access Point Password
	WIFIWEPKey_t	cKey	Wi-Fi WEP Keys
	WIFIWPAPassphrase_t	cPassphras	Wi-Fi WPA Pass Phrase
SmartThings SDK	iot_wifi_prov_data	password	Wi-Fi Password
	iot_wifi_conf	pass	Wi-Fi Password
	iot_security_pk_params	seckey	Private Key
	iot_security_cipher_params	key	Shared Key
	iot_security_ecdh_params	t_seckey	Things Secret Key
	iot_net_connection	key	Private Key
	iot_security_buffer	p	Generic Sensitive Data
Google IoT SDK	iotc_connection_data_s	password	MQTT Client Password
	iotc_crypto_key_data_t	crypto_key_union	Public/Private Key
	connect	password	MQTT Password

Table 2: Sensitive data structure fields defined within the IoT frameworks.

we were not able to execute our queries for other ports of the framework. However, we decided to analyze other implementations of `WIFI_ConnectAP` and `WIFI_ConfigureAP` API functions and see whether they had similar copying behavior. Table 3 represents our findings by manually checking the code in the publicly available ports of Amazon FreeRTOS. It is interesting to see that for the `WIFI_ConnectAP` API function out of 12 ports only two of them did not have the problem of leaking sensitive data to memory, which can later be exfiltrated by attackers via exploiting a memory overflow read as in the case of the HeartBleed vulnerability. Although leakage of sensitive data within the `WIFI_ConfigureAP` function is not as common as in the case of `WIFI_ConnectAP`, this is, however, mostly due to the implementations of this function being merely dummy functions that return without actually performing the configuration. We think that sensitive data must always be erased from a buffer once the buffer is no longer used and developers should apply this secure programming practice to minimize the attack surface.

4.1.5 API Misuse. We have analyzed the IoT frameworks both for conformance to the the mbedTLS API rule regarding not using the corrupted SSL context when the handshake fails as well as for the SDK API rules we have devised based on the available documentation, e.g., code comments and test cases. Table 5 shows the number of queries for each component of the IoT frameworks. As the table shows, most of the API belong to component classes common across the IoT frameworks such as MQTT, HTTP, WIFI, TLS, and Sockets.

mbedTLS API Rule Checking. We have formulated one query to check if `mbedtls_ssl_read` gets called when `mbedtls_ssl_handshake` fails and another to check the same for `mbedtls_ssl_write` using Query Type 4. We have found violations of these rules. However, it turns out that the violation is within the mbedTLS library rather than any of the IoT framework clients. The violation occurs inside the `mbedtls_ssl_read` function, which may continue performing the handshake process if it was incomplete. The code snippet that violates the API rule is shown in Figure 11. According to the condition at lines 3-4, the function terminates with an error if handshake is not successful and the failure is not due to renegotiation. So, the function continues executing if either the handshake is successful or it fails due to renegotiation. As shown in Figure 2 that shows the official documentation from mbedTLS, using the SSL context, e.g., by `mbedtls_ssl_read_record`[4], ,should be an error

[4]IFLOW supports both exact matching and substring matching when searching for function names. We have used substring mode to deal with function name transformations that typically happen in LLVM bitcode. This is why we were able to detect the `mbedtls_ssl_read_record` as a sink callsite.

Board	Vendor	WIFI_ConnectAP		WIFI_ConfigureAP	
		Copies to local buffer?	Clears sensitive data?	Copies to local buffer?	Clears sensitive data?
stm32l475_discovery	ST	Yes	No	Yes	No
CYW943907AEVAL1F	Cypress	No	NA	No	NA
CY8CKIT_064S0S2_4343W	Cypress	Yes	No	No	NA
CYW954907AEVAL1F	Cypress	No	NA	No	NA
mw300_rd	Marwell	Yes	No	No	NA
cc3220_launchpad	TI	Yes	No	Yes	No
xmc4800_plus_optiga_trust_x	Infineon	Yes	No	No	NA
xmc4800_iotkit	Infineon	Yes	No	No	NA
esp32	Espressif	Yes	No	No	NA
mt7697hx-dev-kit	MediaTek	Yes	No	Yes	No
numaker_iot_m487_wifi	Nuvoton	Yes	No	No	NA
lpc54018iotmodule	NXP	Yes	No	No	NA

Table 3: Amazon FreeRTOS port specific `WIFI_ConnectAP` and `WIFI_ConfigureAP` behavior with respect to copying Wi-Fi password to a local buffer and clearing it.

Framework	Source			Sink			Solution			Time (s)		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Amazon Free RTOS	0	11	2.11	6	82507	29538.57	0	3	0.10	9	19	12.26
SmartThings SDK	0	176	36.04	271	13728	4284.38	0	37	1.19	532	924	565.83
Google IoT SDK	3	15	11.00	0	196	54.86	0	1	0.10	17	31	27.19

Table 4: Leakage of sensitive data.

as the renegotiation case is not mentioned. However, it is also possible that this may be due to an incomplete specification. Either way, this shows that IFLOW is effective in checking API conformance of real-world code. We have disclosed our finding to the mbedTLS developers and they are investigating the issue. IFLOW could also detect this API misuse using Query Type 2, which checks for a direct data-flow from the source to the sink. However, in general Query Type 4, for which we reported the results in Table 6, is more general than Query Type 2 as it can detect the pattern even if there are no data-flows from the source to the sink.

```
if( ssl->state != MBEDTLS_SSL_HANDSHAKE_OVER ) {
  ret = mbedtls_ssl_handshake( ssl );
  if( ret != MBEDTLS_ERR_SSL_WAITING_SERVER_HELLO_RENEGO &&
      ret != 0 ) {
      MBEDTLS_SSL_DEBUG_RET( 1, "mbedtls_ssl_handshake", ret );
      return( ret );
  }
}
//ret == MBEDTLS_ERR_SSL_WAITING_SERVER_HELLO_RENEGO || ret == 0
//keeps using SSL context
if( ( ret = mbedtls_ssl_read_record( ssl, 1 ) ) != 0 )
```

Figure 11: Code within mbedTLS that violates the documented `mbedtls_ssl_handshake` rule.

IoT Framework API Rule Checking. Table 7 shows the results of checking the API rules we devised for the IoT frameworks. In addition to the source, sink, and solution sizes, we also present the total number of queries, T, the number of valid queries, V, with non-zero source and non-zero sink pairs, the number of queries that pass, P, i.e., conform to the API rule, and the number of queries that fail, F. As shown in the table, not all queries were valid either due to the code not compiled into the LLVM bitcode because of the configuration setting or having zero instances for the sink or the source. The latter happens when the API function does not have any callsites. This is due to not having any example code exercising the API function in the compiled code, which is possibly due to not being able to compile most of the port specific implementations. All valid queries pass for SmartThings SDK and Google IoT.

```
ret = _wifiConnectAccessPoint();
...
if( WIFI_IsConnected( NULL ) == pdTRUE ) {
  if( WIFI_Disconnect() != eWiFiSuccess ) ...
```

Figure 12: Sample API with implicit sanitizers.

For Amazon FreeRTOS, we have all valid but four queries pass. The four failed queries turn out to be false positives due to using other forms of sanitizers in the form of checking some program variable that does not directly have a data-flow from the return value of the sanitizing function.An example is given in Figure 12, where the return value of the `_wifiConnectAccessPoint` (sanitizer and the callsite for the source argument) function is not directly used for deciding whether the sink function `WIFI_Disconnect` should be executed. Instead, the connection status is checked using the `WIFI_IsConnected` function. We think that taint analysis, in general, is not suitable for detecting such cases, which can be handled more effectively using abstract interpretation or symbolic execution approaches.

Framework	Component	#Queries
Amazon FreeRTOS SDK	FreeRTOS	22
	MQTT	12
	Sockets	6
	TLS	3
	HTTP	3
	GGD	3
	JSON	3
	WIFI	2
	BLE-MQTT	2
	Task	1
	Total	57
SmartThings SDK	MQTT	20
	Security	16
	Generic Net.	5
	NVRam	5
	Http	4
	WIFI	4
	File Sys.	4
	JSON Web Tokens	1
	Total	59
Google IoT SDK	MQTT	16
	TLS	9
	Generic Net.	8
	File Sys.	7
	Sockets	5
	Resource Man.	4
	Control Topic	4
	JSON Web Tokens	1
	Total	54
	Overall	172

Table 5: Various API Misuse queries.

4.2 Optimizations

Table 8 shows query solving results for different reachability bounds of 5 (B5), 10 (B10), 50 (B50), and the case of unbounded. For the sensitive leak queries, the unbounded case shows similar performance to those of the bounded cases. However, for mbedTLS API misuse query for `mbedtls_ssl_read`, unbounded did not terminate in 6 hours. Although bounded analysis may miss some cases, it enables scaling the analysis to real-world code in general.

Figure 9 shows the impact of summarization on solving the mbedTLS API misuse queries, which may have either `mbedtls_ssl_read` or `mbedtls_ssl_write` as the sink. In these specific queries that we analyzed we saw up to a 29% reduction in solving the actual Type 4 query. The reduction in the total time, which includes structural queries that identifies the source and the sink entities, is more modest as the function summaries are not relevant to such structural queries.

5 DISCUSSION & LIMITATIONS

In some cases, we found out that the API rule we constructed was wrong. As an example, in Amazon FreeRTOS, one of the API rules we define states that `SecureSocketsTransport_Disconnect` should be called only if `SecureSocketsTransport_Connect` was successful. IFLOW has detected a violation of this rule within the `discoverGreengrassCore` function. Although the comment in the code acknowledges the fact that the `SecureSocketsTransport_Disconnect` call is performed deliberately in the case of a failure, it turns out that the `establishConnect` function, which gets called from the `SecureSocketsTransport_Connect` function, closes the socket when there is a failure. So, in the case of a connection failure, the socket gets closed using the `SOCKETS_Close` function the second time from the `SecureSocketsTransport_Disconnect` function. The `SOCKETS_Close` function first checks if the socket is valid and not in use before it performs some free operations, e.g., the server's certificate. So, we think that it is probably safe to call the `SecureSocketsTransport_Disconnect` function even if the `SecureSocketsTransport_Connect` fails. However, when we checked the tests in Amazon FreeRTOS, we realized that there were no integration tests that analyzed this case as the existing tests assumed that the connection could be established successfully. We think that API rule coverage can be a useful metric for assessing quality of integration tests.

IFLOW is effective in formulating API misuses and data-flow involving specific data types. However, IFLOW is not suitable for detecting vulnerabilities such as memory overflows or any query that requires precise semantic analysis. We think that IFLOW can be combined with more precise analysis approaches such as software model checking and symbolic execution in a way that IFLOW can predict suspicious code locations, which can get analyzed further by these more precise analyses for the realizability of the memory vulnerabilities.

We were only able to compile Amazon FreeRTOS with clang for the STM 32l475 board. Even for this case we had to deal with some assembly code. Although some vendors have already incorporated the clang compiler into their tool chains, in the IoT world this adoption is still taking place slowly. Although binary analysis is an alternative way to deal with the problem, when source code is available, analysis at the IR level has advantages including the ability to leverage existing source-level mature program analysis tools such as SVF. We urge more vendors to support the clang compiler so that tools like IFLOW can be incorporated to the development workflow and assist developers in finding security issues early on.

6 RELATED WORK

Static Taint Tracking. In [13], reachability analysis on source-sink pairs are followed by data-flow analysis to detect leakage of sensitive data in IOS app binaries.

FlowDroid [9] implements static data-flow analysis based on the Interprocedural Finite Subset (IFDS) framework [20] for Android Apps. It computes method summaries to scale the analysis. However, it does not support sanitization. Phasar [22] is an LLVM-based static analysis framework that also supports the IFDS framework, which can be used to implement taint analysis. However, Phasar uses LLVM's points-to information, which is less precise than SVF's inter-procedural points-to analysis [22]. DR.CHECKER [18] is a static analyzer designed for analyzing Linux device drivers and includes limited forms of taint tracking, where the sources are the arguments of the entry functions or those of special kernel functions. IFLOW computes function summaries based on the Sparse

Framework	Source			Sink			Solution			#P	#F	Time (s)		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg			Min	Max	Avg
Amazon Free RTOS	1094	1094	1094	934	1162	1071	0	96	48	1	1	55	88	71.50
SmartThings SDK	787	787	787	1129	1129	1129	19	52	35.50	1	1	926	967	946.50
Google IoT SDK	122	122	122	115	128	122	0	68	34	1	1	63	64	63.50

Table 6: mbedTLS API misuse.

Framework	Source			Sink			Solution			#T	#V	#P	#F	Time (s)		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg					Min	Max	Avg
Amazon Free RTOS	0	1358	339.01	0	1413	274.54	0	241	5.80	57	24	20	4	3	2874	91.22
SmartThings SDK	0	515	98.75	0	1002	95.89	0	0	0	59	32	32	0	862	7986	1362.12
Google IoT SDK	0	9	1.79	0	16	2.83	0	0	0	54	20	20	0	30	39	32.71

Table 7: IoT framework SDK API misuse.

Framework	Query Type	B5		B10		B50		Unbounded	
		#Sol	Time (s)	#Sol	Time (s)	#Sol	Time (s)	#Sol	Time (s)
Amazon FreeRTOS SDK	Type 1	3	7	3	9	3	9	3	9
SmartThings SDK	Type 1	20	454	31	463	37	463	37	469
Google IoT SDK	Type 1	0	17	1	17	1	17	1	17
Amazon FreeRTOS SDK	Type 4	62	91	62	90	96	88	-	-
SmartThings SDK	Type 4	52	928	52	926	52	935	-	-
Google IoT SDK	Type 4	46	60	46	60	68	65	-	-

Table 8: The impact of the bound on the query solution size and solving time for Query Type 1 (sensitive data field leak) and Query Type 2 (APi misuse with sanitization). – means timeout after running for 6 hours.

Framework	Sink	Type 4 Time (s)		Total Time (s)	
		-Summary	+Summary	-Summary	+Summary
Amazon FreeRTOS	read	115	82	119	86
	write	68	49	73	53
SmartThings SDK	read	418	377	1072	995
	write	446	439	1162	1133
Google IoT SDK	read	38	34	66	60
	write	38	34	64	60

Table 9: Impact of summarization on query solving time for the mbedTLS API misuse query.

Value Flow Graph and it implements a variety of query types including return value based and conditional sanitization.

Weighted Push Down Systems (WPDS) allows formulating regular expression queries for specifying paths of interest [21]. IFLOW allows formulation of several data-flow queries in the form of source-sink relationship and sanitization constraints.

Static Taint Analysis is customized in [25] to scale it for the analysis of production code through lazy call graph construction, inter-procedural def-use analysis, and parallelization based on the sources. IFLOW implements bounded search and data-flow summaries to improve scalability.

AndroidLeaks uses static taint analysis to detect sensitive leaks such as WIFI state information in Android apps [16]. TAJ [24] combines flow-insensitive data-flow propagation over the heap with flow and context sensitive data-flow propagation over the local variables to perform taint analysis for web applications.

IoT Security Analysis. Previous work on data-flow analysis in the context of IoT focused on the sensitive data leaks from the IoT apps [10, 11, 15, 26]. FlowFence [15] performs dynamic taint tracking to enable an information-flow enforcement framework for IoT apps, which are executed in a sandbox to monitor sensitive data-flows. ProvThings [26] generates provenance data to facilitate attack analysis. It uses static data-flow analysis to optimize code instrumentation. IoTWATCH [10] uses NLP and static data-flow analysis to instrument code that process sensitive data and detect run-time privacy violations. IFLOW can support dynamic analysis approaches like ProvThings, FlowFence, and IoTWATCH during code instrumentation and dynamic taint propagation and help extend their analyses to information-flow tracking on the IoT devices and hubs.

Sensitive leaks are detected in commodity IoT apps in [11] using static analysis. Taint tracking is performed between five types of sources (device state, device information, location, user inputs, and persistent state variables) and two types of sinks (internet and

messaging services). This work is complementary to our work as our approach targets sensitive data-flows and API misuse in IoT frameworks.

Firmware rehosting and fuzzing approaches [12, 14, 27] can detect vulnerabilities in IoT frameworks. IFLOW can support these approaches by guiding fuzzing to the parts of the code that hosts specific data-flows.

In [19], static taint analysis has been used to detect insecure code patterns in Industrial Robot Programs that are implemented using a Domain Specific Language. However, the approach in [19] supports a simple form of sanitization: existence of a function call whereas IFLOW uses a return value constraint.

7 CONCLUSIONS

We have presented the first static taint analysis tool, IFLOW, that can apply sanitization rules based on the API return values. IFLOW can also use accesses to specific data fields of data structure types as sources and sinks. IFLOW fills an important gap for the analysis of system code implemented in C. We have shown effectiveness of IFLOW by applying it to three popular open-source IoT Framework SDKs. IFLOW is effective in formulating API misuse queries and sensitive data-flow leakages. Our results show that IFLOW is effective in finding API misuses in real-world code and in identifying vulnerable code locations such as those that copy sensitive data to local buffers. We think that tools like IFLOW can support secure evolution of IoT frameworks. In future work, we are planning to integrate IFLOW with more precise yet less scalable analyses such as symbolic execution to check for memory vulnerabilities within components that depict certain suspicious code patterns.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback. This work was funded by the US National Science Foundation under the CNS-1942235 award.

REFERENCES

[1] [n.d.]. . "https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/". last accessed September 2021.
[2] [n.d.]. Amazon Alexa security bug allowed access to voice history. "https://www.bbc.com/news/technology-53770778". last accessed September 2021.
[3] [n.d.]. Amazon FreeRTOS. "https://github.com/aws/amazon-freertos". last accessed June 2021.
[4] [n.d.]. ambiot-sdk. "https://github.com/ambiot/ambd_sdk". last accessed September 2021.
[5] [n.d.]. DDoS attack that disrupted internet was largest of its kind in history, experts sa. last accessed September 2021.
[6] [n.d.]. Google Cloud IoT Device SDK for Embedded C. "https://github.com/GoogleCloudPlatform/iot-device-sdk-embedded-c". last accessed June 2021.
[7] [n.d.]. SmartThings SDK for Direct Connected Devices for C. "https://github.com/SmartThingsCommunity/st-device-sdk-c". last accessed June 2021.
[8] [n.d.]. "BrickerBot" Results In Permanent Denial-of-Service. "https://www.radware.com/security/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/". last accessed September 2021.
[9] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 259–269.
[10] Leonardo Babun, Z. Berkay Celik, Patrick D. McDaniel, and A. Selcuk Uluagac. 2021. Real-time Analysis of Privacy-(un)aware IoT Applications. *Proc. Priv. Enhancing Technol.* 2021, 1 (2021), 145–166.
[11] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1687–1704.
[12] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*.
[13] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society.
[14] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1237–1254.
[15] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 531–548.
[16] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7344)*, Stefan Katzenbeisser, Edgar R. Weippl, L. Jean Camp, Melanie Volkamer, Mike K. Reiter, and Xinwen Zhang (Eds.). Springer, 291–307.
[17] Susan Horwitz and Thomas W. Reps. 1992. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992*, Tony Montgomery, Lori A. Clarke, and Carlo Ghezzi (Eds.). ACM Press, 392–411.
[18] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1007–1024.
[19] Marcello Pogliani, Federico Maggi, Marco Balduzzi, Davide Quarta, and Stefano Zanero. 2020. Detecting Insecure Code Patterns in Industrial Robot Programs. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. 759–771.
[20] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61.
[21] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.* 58, 1-2 (2005), 206–263.
[22] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 393–410.
[23] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266.
[24] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 87–97.
[25] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. 2020. Scaling Static Taint Analysis to Industrial SOA Applications: A Case Study at Alibaba *(ESEC/FSE 2020)*. 1477–1486.
[26] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl A. Gunter. 2018. Fear and Logging in the Internet of Things. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.
[27] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/zhou