RICE UNIVERSITY

Finding Tizen Security Bugs Through Whole-System Static Analysis

by

Daniel Song

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Master of Science

APPROVED, THESIS COMMITTEE:

Dan S. Wallach, Chair
Professor of Computer Science

Vivek Sarkar
Professor of Computer Science

Lin Zhong
Professor of Electrical and Computer Engineering and Computer Science

Houston, Texas
April, 2016
ABSTRACT

Finding Tizen Security Bugs Through Whole-System Static Analysis

by

Daniel Song

Tizen is a new Linux-based open source platform for consumer devices including smartphones, televisions, vehicles, wearables, and eventually other IoT devices. While Tizen provides kernel-level mandatory policy enforcement, it has a large collection of libraries, implemented in a mix of C and C++, which make their own security checks, raising concerns if any checks are missing or incomplete. In this research, we describe the design and engineering of a static analysis engine which drives a control flow analysis for the full library stack. We implemented the static analysis as an extension to LLVM, requiring us to improve LLVM’s native analysis features with respect to precision and scalability. Our extended static analysis handles knotty issues like the coexistence of C++ inheritance with C function pointer use. With our tools, we found several unexpected behaviors in the Tizen system, including several missing security checks which can be directly exploited by malicious apps, demonstrating the importance of automated checking. We believe that our approach will be applicable to future platforms for the emerging Internet of Things (IoT) which native code is a necessity for many consumer devices.
## Contents

Abstract ii  
List of Illustrations v  
List of Tables vi  

1 **Introduction** 1  

2 **Background** 4  
   2.1 Tizen Applications 4  
   2.2 Tizen Privileges 5  
   2.3 SMACK 6  

3 **Static Analysis** 8  
   3.1 Structure of Analysis Engine 8  
   3.2 Static Analysis Techniques 10  
      3.2.1 Class Type Analysis 11  
      3.2.2 Heap Static-Single Assignment Form 12  
      3.2.3 Call Graph Construction 13  
      3.2.4 Handling C/C++ Features 16  
      3.2.5 Soundness of Static Analysis 18  

4 **API Analysis** 19  
   4.1 Comparison with Standard LLVM Call Graph Analysis 21  
   4.2 Tizen API Analysis Evaluation 22  
      4.2.1 Tizen API Analysis Results 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2 Evaluation of the static analysis</td>
<td>23</td>
</tr>
<tr>
<td>5 Pragmatic Issues</td>
<td>25</td>
</tr>
<tr>
<td>6 Assertion Guided Bug Finding</td>
<td>27</td>
</tr>
<tr>
<td>7 Related Work</td>
<td>30</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>33</td>
</tr>
<tr>
<td>Bibliography</td>
<td>35</td>
</tr>
</tbody>
</table>
# Illustrations

2.1 Tizen application development stack. In this paper, we focus on the native applications stack. ........................................... 5

3.1 Basic LLVM workflow ....................................................... 8
3.2 Internal workflow for LLVM-based analysis ................................. 9
3.3 Class type analysis example .................................................. 11
3.4 Main loop, call graph analysis ................................................. 14
3.5 Call graph analysis algorithm ................................................ 18

4.1 Example of Tizen API Analysis ............................................. 20
4.2 Call graph edge that cannot be detected with default LLVM analysis. .... 21

6.1 Call graph including assertion .............................................. 28
6.2 Code leading to bug CVE-2015-6766 ..................................... 28
4.1 Manual analysis of the tool outputs, including false positives. (There were no false negatives.)
Chapter 1

Introduction

Static analysis has proven to be wildly successful in finding all sorts of bugs, whether related to security or other flaws, so the availability of a new system to analyze for bugs is an interesting opportunity to see how good these tools can be. To that end, we had the opportunity to design and implement static analyses for Tizen, a new operating system platform that will soon run on a variety of products including televisions, wearables, automobile telematics systems, smartphones, and other future IoT devices. This paper describes the analysis challenges presented by the Tizen platform, as distinct from competing platforms like Android, along with the tools we developed and the issues we found.

We’ll describe the Tizen architecture in more detail later, but at a high level Tizen is a variant of Linux, with kernel-enforced mandatory access control rules. Applications can be built entirely from HTML5 web primitives (JavaScript, etc.), much as was done in Palm’s WebOS, or they can be built natively, using a variety of C and C++ standard libraries. Tizen has a series of permissions that can be granted to applications in a fashion similar to Android, which are then enforced both at a low-level, using the kernel, along with higher-level checks embedded in the libraries.

Tizen native apps will be distributed as LLVM bitcode—a portable, machine-independent intermediate code representation that’s naturally amenable to static analysis via the LLVM toolchain. We presume there will be a centralized Tizen app store—Samsung just opened TizenStore.com in January of this year—that can conduct analyses over Tizen apps to ensure their safety prior to being downloaded to Tizen users. In a recent talk, Samsung’s...
partner, AhnLabs, described a mixed process for validating candidate applications for Tizen App Store with both static and dynamic analysis as well as human analysts [1,35], with most of its emphasis on the dynamic analysis.

In deciding what aspects of the Tizen system were interesting for a security-related static analyses, we decided to focus our attention on higher-level security concerns. The Tizen system libraries, written in a mix of C and C++ and containing internal security checks that make them part of the system’s sizable trusted computing base, pose a challenge beyond identifying security vulnerabilities in Tizen applications. These libraries enforce security properties while they are simultaneously linked to the same address space as the potentially hostile apps that call them. We consequently expect that the Tizen app store will need to statically analyze apps to ensure they only branch to approved entry points in the system libraries and that they don’t exploit unsafe properties of the C language (e.g., indexing beyond the end of an array, overwriting a function pointer, and branching to a forbidden target). Such “safety” analyses are well within the province of existing commercial tools, so we didn’t implement them. Furthermore, apps built using the web stack (JavaScript, etc.) call into the very same libraries, pointing to the importance of validating these entry points’ use of security checks.

Consequently, we decided to implement an interprocedural control flow analysis over the native libraries in order to discover whether there are paths through the libraries that are missing security checks, and thus might indicate exploitable flaws that such a “safety” analysis in the app store might otherwise approve. We envision this Tizen library analysis to be something that can run as needed for hours, if not days, in the service of Tizen system developers’ internal bug finding. Likewise, we envision that Tizen system developers would be able to add trusted code annotations to further tune the analysis if needed, although it’s essential that such annotations be few and far between, in order to minimize
friction to the adoption of our tool.

LLVM’s built-in static analysis machinery is very good, but still not good enough for our needs. Consequently, to accomplish our security analysis objectives, our static analysis contribution is a new scalable algorithm for flow-sensitive and field-sensitive interprocedural C++ class analysis, which includes the use of heap SSA form for a uniform treatment of struct/class fields and array accesses; and a call graph construction algorithm which integrates the handling of virtual function function calls and function pointers.
Chapter 2

Background

The Tizen platform is an operating system based on the Linux kernel and the GNU standard C library. It includes a graphics layer based on the Enlightenment Foundation Libraries and the X Window System. Tizen already runs on smartphones, watches, cameras, vehicle infotainment systems, TVs, and in the future refrigerators, air conditioners and washing machines. Consequently, its security properties become quite important.

The Tizen libraries are implemented as a C++ layer of programmer-accessible APIs built on top of a C layer of APIs that are deliberately hidden from the application programmers. The intention is that that application programmers won’t deal, for example, with the X Window System, but rather will use Tizen’s official graphics APIs.

2.1 Tizen Applications

Applications can either be based on HTML5 or native apps. This paper focuses on security analysis of the native applications, which use the C standard library and additional Tizen APIs that offer access to phone calling and contacts, SMS, networking, Bluetooth, and other services as shown in Figure 2.1.

The availability and wide range of these APIs makes Tizen a unique target for analysis since the entirety of kernel, standard libraries, standardized application platform and the applications themselves are written in a mix of C and C++ and compiled to native code.

*Native apps will actually be distributed as LLVM bitcode, compiled at install-time, much like Android*
Figure 2.1: Tizen application development stack. In this paper, we focus on the native applications stack.

These libraries are, in effect, all within the trusted computing base of the platform.

2.2 Tizen Privileges

The main mechanism for enforcing privacy and security for applications is a system of privileges, functionally similar to that of Android. The application privileges are displayed to the user ahead of installation, with applications only being downloaded and installed once the user accepts the privileges that the application requires.

From a security standpoint, the use of C/C++ for the Tizen libraries—widely known as difficult to analyze with its use of function pointers, aliased arrays and deep class hierarchies—together with the existence of rich application APIs, each with their own associated permissions, makes determining the correctness of Tizen’s privilege system a serious challenge. Even for Android, where privileges are enforced outside of a poten-
tially hostile application’s address space, researchers have discovered multiple permissions inconsistencies inside the OS libraries [17] and several different types of permission misconfiguration [17, 32], leading to application over-privilege [38] and increased application vulnerability [2, 34].

While, to the best of our knowledge, Tizen does not have a public available security document explaining the rules of privilege enforcement, by analyzing the code, we observed the following rules.

- An access controller invoked by each privileged API denies access to the native APIs for which an application does not have the privilege. This is done by including a call to CheckPrivilege(privilege_name).

- Third, since checks done in the application process may be avoided by an attacker, protected actions are performed or information is retrieved from other service processes, which perform their own checking for permissions.

- At the bottom level, the inter-process communication and data access is protected by a kernel-level security module (SMACK), described below.

On its surface, this appears to be an example of defense in depth, i.e., perhaps the higher-layer checks are unnecessary and SMACK can carry all the security burden, but we hypothesize that the checks at each layer are necessary, as higher-level API semantics may be lost when control flow reaches the system-call boundary. SMACK appears not to have adequate context to make every security decision correctly on its own.

2.3 SMACK

Simplified Mandatory Access Control Kernel (SMACK) is a Linux kernel module and associated utilities that allow setting custom mandatory access control (MAC) rules to protect
data and limit process interaction. Much like SELinux, SMACK allows a series of rules to be defined about which subjects (e.g., users and programs) are allowed to interact with which objects (e.g., files, IPC services, and system calls). We note that SMACK’s security policies, just like the SELinux policies used in Android, are helpful in setting up filesystem restrictions, but don’t map one-to-one with the privileges that apps request. Those are enforced separately.

We note that the SELinux policy for Linux 2.4.19 consists of over 50,000 policy statements, including over 700 subject types and 100,000 permission assignments [36]. While Tizen’s SMACK is simpler than SELinux, Tizen 2.1 has 41,000 lines of SMACK access rules [33]. It’s manifestly unclear whether these rules are “correct” or how to even define correctness over them. For the purposes of this study, SMACK is largely orthogonal from the analysis we wish to conduct. We won’t be finding errors in Tizen’s SMACK policies, and SMACK won’t be able to catch the vulnerabilities we hope to find in Tizen.
Chapter 3

Static Analysis

The motivation for this work is to identify security bugs in a C/C++ code base through static analysis. The code base could be a mobile application (i.e., a Tizen app) or an operating system (i.e., Tizen), but this paper will focus on the Tizen operating system. We built our analysis infrastructure on top of the LLVM framework. Figure 3.1 shows the basic flow of our analysis system. The C/C++ code is compiled and translated to LLVM bitcode by the Clang compiler. The bitcode is input to the LLVM-based analysis engine, where we can define analysis rules that will drive the engine.

![Figure 3.1: Basic LLVM workflow](image)

3.1 Structure of Analysis Engine

Figure 3.2 shows the structure of our analysis engine, which is built on the LLVM framework (the bold boxes indicate components that we have added). The engine takes LLVM
bitcode as input and translates it into an in-memory LLVM intermediate representation (a three-address static-single assignment based IR). A client static analysis (Tizen API analysis in this paper) runs on the LLVM IR and identifies security bugs. To assist the client analysis, a series of auxiliary analyses are invoked to create additional in-memory information, including the heap static-single assignment (HSSA) form (more detail is provided in Section 3.2), class hierarchy information, class type information and the call graph. The “refined in-memory LLVM IR” is the in-memory LLVM IR augmented by this additional information.

Figure 3.2: Internal workflow for LLVM-based analysis
Here we summarize the functionality of the auxiliary analyses & transformations, and the interactions between them:

- **Class Hierarchy Analysis (CHA):** builds the class hierarchy tree for C++ code;
- **HSSA builder:** constructs the HSSA form;
- **Pointer Analysis (PTA):** intra-procedural pointer analysis;
- **Global Value Numbering (GVN):** global value numbering based on PTA;
- **Class Type Analysis (CTA):** a flow-sensitive class type analysis that is based on CHA, HSSA, PTA and GVN;
- **Call Graph Analysis (CGA):** the call graph construction based on CTA which can precisely identify the invoked virtual function calls, including function pointer invocations.

### 3.2 Static Analysis Techniques

This section gives a more detailed description of the functionality of the auxiliary analyses and transformations. The pointer analysis (PTA) and global value numbering (GVN) are standard LLVM analysis modules. The pointer analysis is an intra-procedural stateless analysis that uses allocation sites to distinguish memory addresses. The global value numbering uses alias information produced by pointer analysis to number the heap variables that have distinct values.

We assume the reader is familiar with static-single assignment (SSA) as the standard intermediate representation for compilers and analysis tools. For an introduction to the topic, see, e.g., Cooper and Torczon’s compiler textbook [12].
3.2.1 Class Type Analysis

In C++, a virtual method invocation may have a variety of different call targets by virtue of class inheritance. A class type analysis can reduce the number of possibilities. Disambiguation of call targets then reduces false positives in a static security analysis, which must otherwise assume any possible target that might be reached will be reached for each given call site. A traditional code optimizer, if it can find a unique target for a call site, can replace the virtual method invocation with a direct function call. For our security purposes, we don’t require such precision, although more precision implies fewer false positive bug reports.

The first step of class type analysis (CTA) is class hierarchy analysis (CHA), which examines class information to build the tree structure that represents the C++ class hierarchy. Figure 3.3 (a) gives a simple class hierarchy example, where classes B and C are subclasses of class A. The class hierarchy tree is presented in Figure 3.3 (b).

LLVM provides a scalar variable-based SSA form to connect definition-to-use (“def-use”) information for scalar variables. For heap variables, CTA needs assistance from pointer analysis. In Figure 3.3 (c), the example code presents a case where pointer analysis information can disambiguate class types. In Line 5, the value of variable \( m \) is loaded from
which can be an instance of class B or C. To identify the type of variable $m$, we need to know if variables $p$ and $q$ are aliased or not. If from pointer analysis we know that $p$ and $q$ cannot be aliased, then $m$’s class type is B, and the invoked function $foo$ in Line 5 is $B::foo$. Otherwise, both $B::foo$ and $C::foo$ may be invoked at Line 5.

We now describe an interprocedural, flow- and field-sensitive class type analysis that starts from class instantiation sites and propagates class type information via variables’ def-use chains. The def-use information is built upon both scalar SSA (for scalar variables) and heap SSA (for heap variables, see more details in the next section). For each scalar variable defined, all of its uses are checked and their class types are updated. If the use is a merge $\phi$ function, a meet update operation is performed, i.e., merging the class type into the merge $\phi$ function’s class type set. For each heap variable defined, all of its may-alias uses are checked and their class types are updated (i.e., merging the class type into the heap variable’s class type set). The operation of the heap variable’s merge $\phi$ is the same as for scalar variables.

### 3.2.2 Heap Static-Single Assignment Form

Heap SSA (HSSA) form [18] is used to represent the definitions and uses of heap variables, i.e. class/struct field and array accesses in the C/C++ context. For each heap variable definition and use, a pseudo-variable $H_i$ is used to annotate the heap variable access, where a $d\phi$ function is used for definitions and a $u\phi$ function for uses. The $d\phi$ and $u\phi$ functions take the heap address (e.g., $p$) and offset (e.g., the offset of struct info’s field $x$) as input parameters that represent the heap position. Similar to scalar SSA, a merge $\phi$ node is used to merge $d\phi$ or $u\phi$ nodes where control flow edges join. Figure 3.3 (d) shows the transformed HSSA form from Figure 3.3 (c). Two $d\phi$ functions (i.e., $H_1$ and $H_2$) are added to heap definitions at Lines 1 and 3, one $u\phi$ (i.e., $H_4$) is added to a heap use at Line 4, and
a merge $\phi$ node is used to merge $H_1$ and $H_2$.

Recall from the CTA algorithm, that class type information can be propagated via HSSA def-use chains. At Line 1, $H_1$ is assigned class type B. $H_2$ is assigned class type C. $H_1$ propagates its class type information through HSSA def-use chains, and reaches $H_4$ as a use, since $H_1$ and $H_4$ are must aliases. For variable $m$ at Line 4, its class types depend on the type of $p$ and $q$, since the definition of $H_4$ comes from $H_3$ which merges $H_1$ and $H_2$. If $p$ and $q$ may aliases, then $m$ takes class type B and C. If $p$ and $q$ must not alias, then $m$ takes class type B only. Building the HSSA form simplifies the manipulation of heap variables for analysis. The may/must alias checking gets help from pointer analysis or global value numbering (i.e., PTA or GVN in LLVM, respectively).

For function invocations, HSSA connects those call sites whose target functions have a side effect (i.e. a load or store of a heap variable). For example, a $\texttt{u}$ function is assigned to the invocation of function $\texttt{foo}$ at Line 5 in Figure 3.3 (d), since function $\texttt{foo}$ performs a load operation on heap variable $B::val$.

### 3.2.3 Call Graph Construction

Precise call graph analysis (CGA) for C/C++ code depends on precise class type information to identify the targets of virtual function calls, and precise pointer analysis to identify the target function pointers. Our class type analysis and pointer analysis are both flow-sensitive interprocedural analyses, and so are based on the call graph. Thus we have a cyclic dependence between call graph construction and the identification of targets of virtual function and function pointer calls.

Our algorithm for call graph construction handles virtual function and function pointer calls in a single integrated analysis. The algorithm iteratively constructs the call graph based on class type and function pointer analysis of the call sites within functions. In this
paper, an entry function might be one of the Tizen API functions for which permissions are required. We would expect that a permission check follows, and its absence would be indicative of a security flaw. To verify this, a call graph is constructed with that entry function as the root node, discovering all reachable locations. The call sites of each reachable function are analyzed and call graph edges and the function targets of these edges are added as needed. A worklist of the functions that still need to be analyzed is used to implement this iterative algorithm. The worklist is initialized to all entry functions of the analyzed code base. (In other words, we’re doing a breadth-first search over the call graph.)

---

**Function: CGA()**

**Input**: $P$ : input program

**Output**: $CG$ : call graph

$W := \text{GetEntryFunctions}(P)$;

$CG := \emptyset$;

while $W \neq \emptyset$ do

$F := \text{PopFirst}(W)$;

$\text{Analyze}(F, W, CG)$;

return $CG$;

---

**Figure 3.4 : Main loop, call graph analysis**

---

**Call Graph Analysis Algorithm** Figure 3.4 presents the main procedure of the worklist-based call graph analysis (CGA) algorithm. The CGA algorithm takes the given program $P$ as input. It collects the entry functions and uses them to initialize the worklist $W$, then pops functions from the worklist and processes them (via procedure $\text{Analyze}$) until the worklist is empty. The $\text{Analyze}$ procedure performs the per-function based processing; new callee
functions are added to the worklist during this processing.

Figure 3.5 provides a detailed specification of the Analyze procedure used in CGA. The first step (see Line 2) is to perform intra-procedural pointer analysis (i.e. PTA), which identifies points-to information for each variable within the given function F. PTA returns a set of modified HSSA nodes (as described in Section 3.2.2), where each node represents either a normal heap access (e.g. class field or array element) or a function invocation that has a side-effect (i.e. where $u\phi$ and $d\phi$ nodes are used to represent the side-effect in callee functions).

After collecting points-to information, the potential target functions for each call site can be updated. The variables’ points-to sets may contain pointers that have class types; the class type information needs to be propagated through the variables’ def-use chains (see Line 3, ClassTypeProp). This process also returns the set of HSSA nodes whose class type information has been changed during propagation.

The next step is to identify the call targets for each call site $C$ (Lines 5 through 20) in the function. CGA handle 3 cases:

1. **Virtual function call**: CGA retrieves all possible class types (i.e. GetClassTypes in Line 10) for $C$ based on the class type lattice corresponding to $C$’s object reference variable (i.e. GetObjectRef in Line 19), and retrieves the offset value from $C$’s virtual function table; it uses the class types and offset value to identify virtual functions (i.e. GetVirtualFunctions in Line 12). The identified callee functions are added to the function set new_callees;

2. **Function pointer invocation**: CGA retrieves the target functions from the call site $C$’s points-to set (i.e. PTS), and adds them to the callee function set new_callees;

3. **Direct function call**: CGA retrieves the function from the call instruction $C$ and adds
it to the callee function set: new_callees.

After collecting the new callee function set for the given call site $C$, CGA compares it with the old callee function set (collected at Line 6), and computes the difference $\Delta$. If $\Delta$ is not empty, the new call edges are added to the call graph and the functions in $\Delta$ are added to the set functions that will later be added to the worklist.

After collecting new callees, CGA also needs to check if the pointer analysis and class type propagation impact old callees that have been already been added to the call graph. If so these callees need to be reanalyzed. Thus CGA revisits the modified HSSA nodes and adds these callee functions into the functions set (see Line 22 to 25). If there is a HSSA node that has function escaping access, which means that its updated points-to information or class type information can be accessed from the current function’s caller, then CGA adds the current function’s callers into the functions set for reanalysis.

The last step in the Analysis procedure is to add the functions that need to be analyzed or reanalyzed into the worklist (see Line 29).

3.2.4 Handling C/C++ Features

C/C++ has features that pose difficulties for static analysis, such as the coexistence of C++ inheritance with C function pointer use, the coexistence of classes/structs, and array element accesses in the form of offsets from pointers. Tizen uses these features, requiring us to handle them correctly. Our call graph construction algorithm, as described above, integrates the handling of invocations through function pointers with the handling of virtual function calls. Our algorithm extends heap SSA form to represent memory accesses through class/struct field accesses and arrays in a uniform way, whereas the original work in heap SSA only supported Java objects, and was extended for C/C++ objects in this work.
Function: Analyze()

**Input**: $F$: function; $W$: worklist, $CG$: call graph

**Output**: modified call graph

$changed\_nodes \cup = PTA (F)$;

$changed\_nodes \cup = ClassTypeProp (F)$;

$functions := \emptyset$;

**foreach** call site $C$ in $F$ **do**

old_callees := GetCallees ($C$);

new_callees := $\emptyset$;

**if** IsVirtualFunctionCall ($C$) **then**

$r := GetObjectRef (C)$;

$ClsTypes := GetClassTypes (r)$;

$offset := GetVTOffset (C)$;

new_callees := GetVirtualFunctions ($ClsTypes$, $offset$);

**else if** IsFunctionPointerInvocation ($C$) **then**

//Get the functions from current call site's points-to set

new_callees := GetFunctions (PTS ($C$))

**else**

//Call instruction has explicit call target

new_callees := GetCalledFunction ($C$);

$\Delta := new\_callees / old\_callees$;

**if** $\Delta \neq \emptyset$ **then**

//Add new callees into call graph

AddCallEdges ($F$, $\Delta$, $CG$);

$functions \cup = \Delta$;
hasEscapeHeapAccess := false;

declare foreach n in changed_nodes do
    if IsCallSite(n) then
        functions ∪ = n;
    else if IsEscapeAccess(n, F) then
        hasEscapeHeapAccess := true;

if hasEscapeHeapAccess then
    functions ∪ = GetCallers(F, CG);
    W ∪ = functions;
return;

---

Figure 3.5: Call graph analysis algorithm

3.2.5 Soundness of Static Analysis

Our call graph analysis is sound. The results of flow-sensitive class type analysis can only be applied to those LLVM function pointer invocations that are C++ virtual function calls. To guarantee soundness, the targets of the given virtual function call are determined by the possible class types and the class hierarchy tree (see Figure 3.5). For other function pointer invocations, the function targets are determined via pointer analysis.
Chapter 4

API Analysis

We now use the call graph analysis CGA described in Section 3 as a basis for Tizen API analysis (TAA). TAA identifies paths from native Tizen API calls to low-level system (Linux) kernel calls to test for potential violations of user privileges. Each entry function is the root of a call graph which is built by CGA. TAA performs dataflow analysis over a call graph, using information flow to propagate the set of privileges checked along each call path. We then use user-specified rules to specify analysis goals. These rules, hand-derived from the Tizen API documentation, specify rules that the code must enforce:

1. A set of (source, sink) pairs, where each source is a Tizen API call and each sink is a glibc function that does a kernel call.

2. A set of user privilege properties (UPVS) required along all paths from source to sink.

TAA traverses the entire call graph in a top-down manner, starting a new call path trace when a source call is identified. This trace is performed on the call graph by means of HSSA. For each call path in the library code base, the TAA collects the set of user privileges (PVS) exercised along the path and stores the call path into a candidate list when a sink call is identified. The privilege is checked from a CheckUserPrivilege function call but the user can also specify other special function calls for identifying user privileges. A call path is a potential violation of user privilege properties if we discover the privileges required to
get from the source to the sink are less than those required (i.e., if PVS is not a subset of UPVS, we output that path to the analyst).

```c
void ButtonEvent() {
    ...
    x = TizenNativeAPI();
    ...
    CheckUserPriv(PRV_1);
    ..
    evaluate(p);
    ...
}
```

```c
void evaluate(Info* p) {
    ...
    CheckUserPriv(PRV_2);
    ...
    BlueToothOp();
    ...
}
```

```c
void BlueToothOp() {
    ...
    glibcCall(m);
    ...
}
```

**Figure 4.1 : Example of Tizen API Analysis**

Figure 4.1 (a) shows a simplified example where the code base contains a user-specified source: *TizenNativeAPI*, sink: *glibcCall*, and the check privilege function: *CheckUserPriv*. There is a call path from *ButtonEvent* → *evaluate* → *BlueToothOp*. There are two user privileges exercised in this call path: *PRV_1* and *PRV_2*. Figure 4.1 (b) gives the HSSA version of the code (only function based *uphi* nodes need to be considered in this analysis), and the arrow lines show the progress of updating PVS in the HSSA def-use traversal for call path.

```c
void ButtonEvent() {
    ...
    x = TizenNativeAPI(); H1 = uphi(H0)
    ...
    CheckUserPriv(PRV_1); H2 = uphi(H1)
    ..
    evaluate(p); H3 = uphi(H2)
    ...
}
```

```c
void evaluate(Info* p) {
    ...
    CheckUserPriv(PRV_2); H11 = uphi(H0)
    ...
    BlueToothOp(); H12 = uphi(H11)
    ...
}
```

```c
void BlueToothOp() {
    ...
    glibcCall(m); H21 = uphi(H0)
    ...
    a.
```

```c
void ButtonEvent() {
    ...
    x = TizenNativeAPI(); H1 = uphi(H0)
    ...
    CheckUserPriv(PRV_1); H2 = uphi(H1)
    ..
    evaluate(p); H3 = uphi(H2)
    ...
}
```

```c
void evaluate(Info* p) {
    ...
    CheckUserPriv(PRV_2); H11 = uphi(H0)
    ...
    BlueToothOp(); H12 = uphi(H11)
    ...
}
```

```c
void BlueToothOp() {
    ...
    glibcCall(m); H21 = uphi(H0)
    ...
    b.
```
The output of TAA is a list of such call paths that potentially violate user privilege properties. The output includes the source Tizen API function, the sink Linux kernel function, and the full call path from source to sink.

4.1 Comparison with Standard LLVM Call Graph Analysis

CGA not only improves the precision of the standard LLVM call graph builder, but also, unlike that builder, constructs the complete call graph. The LLVM call graph builder does not handle call sites that contain constant expressions, and so loses the edges of de-virtualized class function calls. See the `Start` function invocation in Figure 4.2.

```c
result NetConnection::Start(void) {
    result r = E_SUCCESS;
    ...
    r = __pNetConnectionImpl->Start();
    ...
    return r;
}
```

Figure 4.2: Call graph edge that cannot be detected with default LLVM analysis.

The call edge from NetConnection::Start to __pNetConnectionImpl->Start() (which is _NetConnectionImpl::Start) should be present in the call graph but is absent from the call graph built by the LLVM call graph builder. CGA is able to identify such a pattern, guaranteeing the completeness of our security analysis. In building the complete call graph, CGA identified 208 privileged API calls which were not identified by the standard LLVM call graph builder.
4.2 Tizen API Analysis Evaluation

Using our handwritten rules, derived from the Tizen API documentation, we found a number of interesting and unexpected paths through the Tizen codebase.

4.2.1 Tizen API Analysis Results

There are total of 632 privileged APIs in Tizen 2.2 which we analyzed through our tool. We then did a manual evaluation of the results whose results are shown in Table 4.1. Out of the 632 privileged APIs analyzed, the tool identified 517 of the privileged APIs as correctly performing all the required privilege checks. The remaining 115 were labelled as not have adequate privilege checks.

However, not all of them were indeed lacking privilege checks and vulnerable. We dug deeper and followed the subsequent execution paths manually for every privileged API without the privilege check. We have found out that 54 out of 115 APIs performed privilege checks on the IPC end on the server side. This practice has better security properties, since there is no way for the client to somehow branch around these checks. Our analysis does not currently trace across IPC boundaries, although these edges would be straightforward to add.

<table>
<thead>
<tr>
<th>Tool output</th>
<th>Correctness analysis</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negatives (517 APIs)</td>
<td></td>
<td>All checks present as expected (517 APIs)</td>
</tr>
<tr>
<td>Positives (115 APIs)</td>
<td>False positives (115 APIs)</td>
<td>Checks through IPC (54 APIs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capability style checks (40 APIs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Different Privilege Check API (21 APIs)</td>
</tr>
</tbody>
</table>

Table 4.1 : Manual analysis of the tool outputs, including false positives. (There were no false negatives.)
Our analysis also highlighted the `InputMethod` class. None of the `InputMethod`’s 40 privileged APIs had privilege checks, including the `SendText` API. Again, we manually followed the calls and discovered that, unlike other classes’ privilege checks, `InputMethod` enforced privilege checks in its `GetInstance` method, when the application retrieves an instance of `InputMethod`. Assuming references to `InputMethod` classes cannot be acquired in any other way, then security of the system is maintained. The original 1996 Java APIs used for untrusted Java applets did exactly the same thing (e.g., security checks happened when opening network connections, but not when reading or writing to those connections). This code thus resembles a traditional capability-style programming interface (see, e.g., Wallach et al. [37]). Our tool does not presently have the machinery to deal with capability-style objects, but such an extension would be fairly straightforward—basically just metadata on each capability-style class to declare it as such. Because these objects cannot be passed from one address space to another, we don’t have to worry about confinement issues.

At the time of the experiment, `SecureElement’s` 21 methods were using a different version of privilege checking API from everything else. This version used a string as input to name the privilege being checked while all the others used an integer. We believe we can easily extend the tool to handle this alternative API.

### 4.2.2 Evaluation of the static analysis

We tested the static call graph analysis on part of the Tizen platform consisting of 4,346 C/C++ files compiled into LLVM bitcode files with a total size of 560MB. The analysis time for generating the call paths for an API took 122.5 secs with memory usage under 8GB. This is fast enough that it could reasonably run not only as part of a nightly build process but as part of a regular developer’s source code commit process, flagging potential violations before the change hits the code repository.
Our test collection of Tizen code contains 95,703 functions. The CGA algorithm identified 580,452 call edges, which eliminated about 15% of the call edges relative to the default LLVM call graph builder, which identified 682,651 call edges. Recall that a conservative analysis must assume that each call site has an edge to every target function with the correct argument and return types. Also, keep in mind that LLVM’s default call graph builder is already quite powerful and LLVM is used as a production-quality compiler for many real systems, so a 15% improvement is quite valuable.

Also notably, some indirection function call sites have an involved sequence of LLVM bitcode instructions that the LLVM call graph builder does not interpret as a graph edge. Our CGA analysis was able to recognize 10,251 of these call sites and treat them correctly.

Lastly, we note that Tizen doesn’t make heavy use of C++ object features (e.g., it doesn’t have a deep class hierarchy). Other systems, with more complex class structures, would represent a more difficult analysis target, requiring suitably more powerful call graph builders.
Chapter 5

Pragmatic Issues

Our static analysis tool leverages the LLVM analysis infrastructure and so depends on the use of the LLVM compiler. However, the Tizen platform code is natively compiled using GCC. To compile the Tizen platform code with LLVM, we had to address issues that other large-scale static analysis tools, such as Coverity, also must address when processing real-world software, such as the variations in language dialects [13]. In short, to use the LLVM infrastructure, we had to port Tizen to LLVM. Since GCC and Clang are not completely compatible [11], this step involved manual inspection of each module. We edited each build file and made source code changes as needed to remove errors. Changes, in some cases, included editing of assembly code. Furthermore, Tizen uses a variety of different build systems (CMake, libtool, and traditional makefiles). Consequently, each module became a new adventure in software porting.

Consequently, we had to decide when we had enough coverage to validate our tool and approach. The Tizen source is divided into a variety of different source packages; we successfully compiled 159 out of 390 Tizen framework packages to LLVM bitcode, generating more than 4,000 LLVM bitcode files with a total size of 560MB. We included all the packages from the top two layers, OSP and CAPI, which handle native applications. We also ported underlying components’ packages that included a large majority of the Tizen privileged APIs, such as telephone, messaging, and system. We did not compile packages that were not relevant to the privileged APIs such as graphics, UI, and multimedia.

A full analysis, of course, would need to push the entirety of the Tizen codebase through
LLVM, and this effort would need to be replicated each and every time the analysis was to be conducted. If our vision of our tool being closely integrated in the Tizen build environment were to ever take off, Tizen would realistically need to switch to LLVM as its production compiler. With LLVM in production use by a number of very prominent projects, include Apple’s iOS / OS X, this isn’t an unreasonable recommendation.
Chapter 6

Assertion Guided Bug Finding

Chromium  We have further extended our static analysis tool to find bugs in Chromium browser. We chose Chromium, the project behind the Google Chrome browser, not only because it is an open-source project but also has the option to be built using Clang. This remedies all the issues we had with changing the compiler from GCC to Clang for Tizen.

Chromium Security  Chromium uses various techniques such as process separation and process sandboxing to protect itself. One security mechanism that caught our attention was assertions. At source level, Chromium heavily uses assertions to protect from security bugs. Assertions are added when failed assertion would lead to a security vulnerability. We believe that these assertions can be used as guidance to find bugs.

Assertion Guided Bug Finding  We use our tool to find paths to the code region that circumvent the assertion. Assertions are believed to guarantee a set of properties on the certain code region. These properties might not hold for code paths that circumvent the assertion. Figure 6.1 shows an example where a certain code region protected by the assertion can be reached using another code path. Assume function privileged() is called after the assertion(). The code path from foo() to bar() which leads to privileged() after the assertion() holds certain set of properties. However, this property might not hold for the path from foo() to privileged(). We believe we can find this type of paths using our tool which
Another usage of our tool is to find paths to the debug assertions. Chromium uses two types of assertions: debug assertions and release assertions. Due to performance reason, some assertions are removed in the release build. If we can find a path to the debug assertion that fails the debug assertion’s properties, this might also lead to a security vulnerability.

```cpp
PendingMasters::iterator found =
    pending_master_entries_.find(host->pending_master_entry_url());
DCHECK(found != pending_master_entries_.end());
PendingHosts& hosts = found->second;
```

Figure 6.2 : Code leading to bug CVE-2015-6766

Figure 6.2 shows a real world vulnerability [14] which is a use-after-free bug. The vulnerability can be exploited if the attack can reach the last line of code with failing the
property of the DCHECK which is a debug assertion for Chromium. We believe that this can also be detected by further extending our tool.
Chapter 7

Related Work

A large number of other groups, academic and industrial, have built static analysis tools for security purposes. For example, we’ve seen sophisticated tools like TaintDroid [15], FlowDroid [3], DroidSafe [23], and PermissionFlow [32] applying state-of-the-art information flow analysis to Android applications and the Android system library. All of these tools, and many more like them [8, 10, 16, 20, 29, 30], have been able to find real security issues. Our work applies similar static analysis techniques to Tizen. Also of note, many researchers have looked specifically at mapping Android permissions to the code that checks them, either statically [4] or dynamically [17]. We’re taking the former approach in our own work.

Static analysis of production code Static analysis has been proven to be successful in finding bugs in real-world programs. Coverity and Fortify are well-known commercial static analysis tools. An article by Bessey, et al. [13] discusses a number of pragmatic issues and experiences with respect to static analysis tools for finding bugs for large commercial code bases (up to 20-30 MLOC). They observe that "the false positive rate is simplistic since false positives are not all equal and initial reports matter inordinately". Both Fortify and Coverity emphasize results prioritization once vulnerabilities are identified.

IBM AppScan Source is a tool meant to identify bugs during the development phase for web applications. Other editions of IBM AppScan identify general bugs while focusing on security problems in particular and supporting customizable rules.
FindBugs [6], a static analysis tool used on Google code bases, focuses more on identifying common Java programming bugs rather than security vulnerabilities in particular. The importance of the tool’s UI with respect to the speed of understanding and fixing bugs has been demonstrated [5] (analysts processed bugs in FindBugs faster than with Fortify). The tool was used to show that bugs found in older code bases are less likely to be fixed once discovered [7].

ESC/JAVA [19] is a static analysis tool, powered by verification-condition generation and automatic theorem-proving techniques, for Java that checks for common programming errors. While it does find errors, users have to annotate the software and the annotation burden is quite high. It also suffers from excessive spurious warnings on programs that are annotated.

Metal [25] is a language for programmer-written compiler extensions that express a broad range of correctness rules that code must obey. The system xgcc executes these extensions using a context-sensitive interprocedural analysis. Metal is designed for system programmers with an emphasis on ease of use, and makes use of state machines as a fundamental abstraction. This approach has been used to find thousands of bugs in real systems code.

**Interprocedural Class Type Analysis** We discuss the work that is most closely related to ours: call graph construction algorithms that perform interprocedural flow-sensitive class type analysis. Grove, et al. [24] describe a class type analysis and call graph construction algorithm for C++ which they name 0-CFA. The 0-CFA algorithm is similar to ours in that it is fully flow-sensitive and context-insensitive. Their algorithm is field-insensitive: access to a class/struct field, or an element of an array, is treated as an access to the whole object/struct or array. Based on Heap SSA form, our algorithm adds field-sensitivity, distin-
guishing class/struct fields and individual array elements through a uniform representation.

Pande and Ryder [31] present a context-sensitive interprocedural flow-sensitive algorithm for C++ class type analysis for programs with single-level pointers. They introduce context sensitivity through a conditional analysis [28] which takes place in two steps. In the first step, analysis of a single method is performed based on the condition that certain aliases hold at the entry node of a method. A second step performs the analysis based on the actual alias relationships that may hold on entry to the method. Their analysis is field-insensitive.

Chambers and Ungar [9] perform a static interprocedural class type analysis of the dynamic object-oriented language SELF. They refine their analysis by iterative type analysis and extended message splitting. Extended message splitting refines flow-sensitive type information by duplicating code. Iterative type analysis is performed over a changing control flow graph.

**Function Pointer Analysis** The Hind et al. algorithm [27] is the most similar to ours for building the call graph in the presence of function pointers. Their analysis begins with a minimal call graph, i.e., one that only has edges for direct function calls. This call graph is used for a pointer analysis that propagates function pointers to call sites that use them. This allows call graph edges due to indirect function calls to be added. The pointer analysis/call graph construction iterates until convergence. Our call graph analysis algorithm is the first to integrate such a function pointer analysis with class type analysis.
Chapter 8

Conclusion

Analyzing the security of a large software platform like Tizen presents a valuable opportunity to apply state-of-the-art tools in static analysis. Static analysis can be usefully applied to identify undesirable behaviors in apps distributed through app stores, and it can help the system’s developers find needle-in-a-haystack bugs throughout their system.

Our work additionally demonstrates the value of a general-purpose infrastructure like LLVM. While this project focused on C and C++ code, our analyses could potentially run on any programming language for which there’s an LLVM front-end. For example, a JavaScript front-end for LLVM would allow our tools to analyze Tizen “web apps” in addition to “native apps” with identical information flow rules.

Furthermore, the extensions we made to LLVM, such as our class type analysis and precise call graph construction, are general purpose and could well be folded back into the LLVM distribution. (We intend to make an open source release of our extensions.) We hypothesize that the increased precision of our analyses will enable dynamic dispatches to be replaced with static function calls, as well as allowing for better function inlining and other performance benefits. Evaluating this performance impact represents future work.

Likewise, our work discovered that Tizen uses some “capability-style” APIs, where permissions only need to be checked when an object is created or acquired. Creating modest code annotations to assist with this analysis would be straightforward and useful. Furthermore, it would be useful to extend our analysis across IPC boundaries, where it might be useful to discover confused deputy problems [26].
Now that Samsung has shipped its first Tizen products and real apps are starting to appear in its online app store, we expect that independent security analysts will be able to download these apps, in bulk, and analyze them as many security analysts have already done for Android and iOS. As Tizen spreads to more devices, especially IoT devices with more limited resources, we expect that it’s support for efficient native applications will become essential to operating on these devices, bolstering the importance of static analysis to ensure safety and security.
Bibliography


[20] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated secu-


