Abstract

There has been no shortage of Android malware analysis reports recently, but thus far that trend has not been accompanied with an equivalent scale of released public Android application tools. To address this issue, we are presenting the Scalable Tailored Application Analysis Framework (STAAF). The goal of this framework is to allow a team of analysts to efficiently analyze a large number of Android applications and to promote collaborative analysis through shared processing and results. STAAF is designed to be scalable and extensible, and can quickly be customized to meet the current analysis needs. Our hope is that by releasing the framework, we will not only provide an efficient tool for automating and scaling analysis tasks, but we hope the work will encourage the sharing of research in the field.

1. Introduction

Over the past few years, we have seen a revolution in mobile computing. The monumental jump in the capabilities of next generation phones has caused a paradigm shift in the way we view and use our mobile devices. Those changes have also induced an evolving mobile market where old stakeholders are finding themselves in a defensive position against new entrants. In particular, Google and Apple have positioned themselves to dominate the market with their respective Android and iOS mobile platforms. One of the primary success factors behind both Apple’s and Google’s massive consumer adoption is the conscious decision each company made to empower other software providers to enrich their mobile platforms with third party software. The additional functionality these applications provide has drawn an ever greater user base which in turn has drawn even more developers. In this way, each company has created a fanatical and self-reinforcing model. Entire communities have organically sprung up to support the two platforms.

However, the two companies travelled down two different paths to build these communities. Apple maintains a very strong supervisory role in how applications are developed and deployed into their AppStore. Apple requires developers to purchase an annual license that gives a developer the opportunity to publish their applications in their online market, but the application must be vetted and certified by Apple before it’s permitted to be sold in the market. Google on the other hand only requires developers to purchase a license, and after this purchase, they are permitted to publish as many applications as they like without any interference from Google. If the developer violates the Google Android Marketplace’s Terms of Service, the application will be removed from the market, and the developer’s account will be banned. In extreme circumstances, Google will even activate a “kill switch” that will remove these applications from infected Android devices. The remainder of this paper will be focusing solely on the Android platform.

As we have mentioned before, the Android platform and marketplace have been largely successful because it offers users and developers alike a degree of freedom when it comes to creating and using applications and content, but this freedom and extensibility has also left the door open for attackers. Focusing on the Android Middleware, an attacker can create applications that collude with each other as in the prototype trojan SoundComber [1], and Xiang et al. [2] showed how to design a more effective bot that can adequately evade detection while leveraging a number of the device’s features for various covert functionality. Malicious applications can also attempt to escalate their privileges by taking advantage of poorly configured or programmed interfaces [3, 4]. In one case, researchers used formal methods to realize a vulnerability in the way permissions are assigned on the Android at install time, and there has been active and ongoing research in the analysis of permissions and the source code of applications [5].

While breakthroughs have been made in the way analysis is performed on the applications, there has yet to be an attempt to combine these methods into a
formal framework for expedient and efficient analysis. Case in point malware is beginning to become a regular occurrence, and the incidents in which new malware is discovered is growing rapidly over time. There are two problems, which may seemingly be unrelated that really need to be addressed. First, there is no real third party certification of applications to ensure they are trustworthy or that they are clear of currently known vulnerabilities prior to install. Given the fact that third party Android marketplaces can launch at a whim, there should be some service that users and these marketplace owners can submit their applications to in order to ensure they are safe. Currently, the modus operandi is to rely on anti-virus or wait for the compromise to occur, and then wait for the reactive but effective kill switch, which can be issued at the authority of Google. However, Google cannot totally eliminate this threat, because the applications can potentially remain in third party marketplaces. Furthermore, even if the kill switch works for now, we cannot ensure that future malware will not find a way to prevent use of the kill switch.

The second issue that arises is the incident response. What if a forensics team needs to respond to an incident where an Android application is the root cause of the compromise? Current literature only shows the standard practice of rooting the phone and then capturing the data off the memory technology device [6, 7], and no commercial tools seem to address the analysis of Android applications, as potential causes for a compromise, in an automated fashion. Primarily they appear to focus on the simple task of acquiring memory and sifting through device and location logs, resident user data, and artifacts created by the applications.

Our project aims address these two problems by creating a 1) distributed platform for application analysis and 2) methods for performing en masse Android application analysis in a pipelined manner. In doing so, we have developed a modular system that focuses on the dissection and analysis of these applications so that analysts can focus on identifying the root cause of a compromise, potential vulnerabilities, or malicious applications. The analyst can use our system to apply a variety of commodity tools to extract the Dalvik byte code, permissions, and other resources, and this data is then processes and stored. Once this initial data extraction is complete, the analyst can leverage STAAF to process these elements in a distributed fashion using built-in or custom modules.

The layout for the remainder of the paper is as follows. Section 2 will present an overview of past and ongoing research in the area of malware analysis and Android security research. Section 3 describes the major problems to be addressed in more detail, and Section 4 discusses some of the early challenges we faced with our initial system and framework design. Section 5 discusses our proposed framework design and configuration, which specifically addresses the performance issues raised. Section 6 focuses on the result of our system, and Section 7 provides a road map for the future direction of our system. Finally, section 8 concludes this report.

2. Related Work

In much of the initial research, permissions were the focal point of application correlation, primarily because they offer the most insight into what the application is trying to accomplish without performing in-depth analysis. Barrera et al. [8] applied the Self-Organizing Map (SOM) algorithm of Kohonen [9] on 1,100 applications in an effort to try and classify applications using distinct sets of permissions and as a means of trying to determine whether permissions should be made more expressive and granular. Porter et al. [10] focused on mapping permissions to specific actions and activities with their research. To map the permissions, they modified the system responsible for performing verification of application, and then they systematically called APIs using automated and manual interaction to determine which APIs were using the various permissions. These results were used to build a permission mapping, then the mapping was used to create an automated analysis tool, Stowaway, which was used to identify over-privilege in applications. Enck et al [11] performed a security study of 1,100 applications using decompiled application source code. In this work, they created a Dex decompiler, ded, which retargets the software to Java, which in turn was passed to a source code analyzer, Fortify SCA. Their research set out to detect dangerous functionality and vulnerabilities, but they ultimately identified issues related to misconfigurations and misuse of private information.

In the past there has also been research that looked for vulnerabilities or attacks that can be used against Android applications. Shin et al. [5] presented a very subtle flaw in the assignment of permissions to an application at installation time. Through formal methods, they were able to prove that an Android application can request a permission that does not exist, and once the permission is created, any Android components that were previously assigned this permission will be granted access automatically. Chin et al. [4] created
ComDroid, which was built on their permission mapping research [10], and they used this tool to look for components that did not enforce their permissions or had weak permissions in place. Some of the potential attack vectors they noted were spoofing of Android Intents and determining if it was possible to send unauthorized intents to various components (e.g. Activity, Service, Broadcast Receiver) in an application or determine if Intents could be spoofed against the components. In related research, Davi et al. [3] showed how to exploit these vulnerabilities to achieve privilege escalation. Since Android is simply a middleware for Linux, it can also be affected by OS or kernel-level vulnerabilities. Coverity performed a source code scan on the open source portion of Android’s software and found 359 software defects, 25% of which were high risk issues [12].

In a different light, Xiang et al. [2] developed Andbot to get a better understanding of how advanced mobile botnets will operate. Specifically, they investigated how the bot should operate on an Android client to evade detection during installation and execution, and also proposed a sophisticated botnet command and control (C&C). Another proof of concept trojan developed in the academic community is SoundComber [1]. This trojan demonstrated how a few benign permissions and application collusion could be used to build an effective backdoor capable of listening to telephony voice and tones. Once they mined this information, they demonstrated the feasibility of malware that is able to collect coherent and sensitive information (e.g. PINs, account numbers, names) to then covertly steal from the user. Outside of academia, researchers have been identifying malware in the Android markets on a regular basis and at a rate that is steadily increasing. The distributed malware have been found to repackgate legitimate applications [13] or advertise themselves as seemingly innocent applications. There have been several malware that use publically known exploits [13,14,15,16], but this is not a requirement for the malware. Some of the noted functions have been dynamic class loading of libraries [17], sending SMS messages to premium numbers [18,19,20], and the malware is slowly moving towards the use of encryption [16].

TaintDroid [21] is a study that analyzes Android applications through a dynamic lens, and monitors data using taint analysis as it flows through applications and components. The results of this study revealed that applications were exfiltrating private information about the user’s and their phones. QUIRE [22] aims to prevent Intent spoofing along the RPC call chain and ensure the original callee is authorized to make a request. Kirin [23] is an Android application that focuses on certifying applications at install time. Kirin works by parsing the application’s manifest, then comparing the list of permissions and interfaces with a set of rules. When the service completes its analysis, it will return the set of rules and an indication as to whether they passed or failed, then the user can make a more conscious decision as to whether or not they want to install the application. Saint [24] is a run-time extension for the security environment, where applications set policies at install time, and the policies are enforced at the applications’ run-time to control access to their interfaces. This approach improves the access control feature by allowing applications themselves to regulate access based on the RPC caller or callee. XManDroid [25] is a security extension for the Android platform that focuses on extending Android’s reference monitor, which mitigates privilege escalation attacks by colluding applications, the confused deputy attack [26], and covert channels in the ICC. Finally, TISSA [27] introduces additional controls into the Android access control process, which are used to regulate an application’s access to a user’s private information. Furthermore, this extension allows users to provide bogus information to the application, and adjust the privacy settings of the application at any time.

3. Problem Description

As was discussed in earlier sections, there is little, if any, proactive security validation, vulnerability scanning, or malware scanning done by either the official Android marketplace or the many other third party Android marketplaces. Further complicating matters is the fact that Android applications can be hosted and installed through private channels or individual servers. Combine this with the fact that by design Android doesn’t distinguish between functionality provided to third party applications and core applications [29], and we can see a stage set for significant malware propagation using the naive user as a vector. While this vector has been common for PC malware propagation, this problem has yet to be seen for mobile platforms at the same magnitude and with the same level of impact.

Much of the previous research focuses on detection of malware on the Android device, and there are several commercial anti-virus tools available for the Android platform. While it’s prudent to run an anti-virus system on one’s Android device, we believe that it’s just as important to be pro-active in seeking malicious applications before they ever find their way to a user’s device. This type...
of third-party marketplace monitoring is essential to identify new malware instances and trends, before they reach a large number of Android users. While Google has been hesitant to employ self-censorship through internal security scanning, they are extremely quick to remove or suspend malicious applications along with their associated publisher’s accounts, but these problems must first be reported by third party security researchers. Goimg forward, we believe that third party Android security scanning and verification will be an essential component of a thriving Android ecosystem.

Existing research projects and available tools have focused on the analysis of a single or relatively small number of Android application samples. We argue that in order to meet the prolific challenge of scanning multiple Android marketplaces, scalability must be addressed, which to date has not been a primary focus of available research. With the existence of the hundreds of thousands of applications available in the various Android marketplaces, the number of new applications increasing by tens of thousands each month [30], and the unregulated nature of the application markets, it’s no longer enough to design a system to test 10 or even 10,000 applications. In order to meet the security demand, we must create high-performance tools for large scale tests on the order of hundreds of thousands or even millions of samples in a reasonable amount of time.

4. Challenges

There are several challenges that must be addressed in order to proactively and continuously monitor the magnitude of disparate Android applications that currently exist. First, we want to ensure that application analysis can scale to handle several thousands of new Android applications within in a reasonable time. Previous research [8, 11] has only focused on performing analysis on a few hundred to a thousand sampled applications. During our own initial analysis of more than 50,000 applications using an early and non-optimized prototype of our framework, we discovered the magnitude of this performance challenge. For example, our initial framework took nearly two weeks to complete all of the processing modules on the initial sample set of 50,000, then required more than 3 days to process all of the more than 60 Gigabytes of data extracted from the applications. We recognized that this poor performance would not be sufficient to satisfy the demand of a continuous large-scale analysis operation, so we identified several key elements that we thought needed to change in order to address these bottlenecks. One of the acknowledged inefficiencies was processing all of the applications serially, even though they could be processed separately in parallel. Additionally several tasks share common resource requirements, so rather than processing the manifest or byte-code each time, intermediate results could be stored and reused by the multiple analysis modules that require these results. Furthermore, many of the analysis tasks only require read-only access to the intermediate resources, so these tasks could be transformed into a parallel process using multi-threading or distributed execution environments. Finally, many of the applications share common resources such as code (e.g. advertising libraries) that don’t need to be reanalyzed every time an analysis module analyzes applications that share the resource. By addressing these core issues, we believe that the new framework will be able to perform with lower overhead and better utilization of compute power and storage, allowing us to achieve analysis results on our library of 50,000+ applications in a much shorter time frame.

The second challenge that we must address is data sharing and collaboration. Current data sharing initiatives are ad-hoc or non-existent, and this problem leads to a duplication of efforts in addition to the lack of a cohesive strategy among team members. However, it’s not always a simple proposition for an analyst to share results of current or past analysis with team members. We believe that the framework must directly address this challenge of information sharing, from the raw inputs to the final results, including all pertinent intermediate data. By addressing this challenge we believe that our framework will enable collaboration between segmented teams and team members, yielding richer results.

Given these challenges, we resolved to design a framework that will allow organizations to perform Android application reverse engineering and analysis at a very rapid pace, even for large numbers of incoming samples. We plan to achieve this by using a modular framework that allows an analyst to quickly write and deploy analysis tasks.

5. STAAF System Design

In this section we present the Scalable Tailored Application Analysis Framework (STAAF), describe its main components, and finally discuss our current implementation of the framework.

STAAF is designed to allow large scale distributed Android application analysis, and achieves this with aggressive parallelization of analysis tasks, deduplication of processing efforts and
data storage, as well as efficient data storage and recall. Because applications can be processed independently of each other, we are able to distribute the load of processing tasks for each application, which showed a marked improvement over the previous serialized application analysis. Additionally, rather than feeding every individual analysis tool the raw application we extract and process the required resources once, and then we feed the processed results into the analysis tools that require those resources. Furthermore, certain aspects of the application, such as library code (e.g. advertising libraries), and certain resources, are often reused between applications. Rather than analyzing these shared resources multiple times for each application that includes them, many tasks can ignore these shared resources, significantly cutting down on the amount of redundant data processing. Finally, we have designed the system using a distributed noSQL database solution. This database design provides low latency storage and recall, and also allows us to transparently include additional remote third party analysis databases for collaborative analysis and data sharing.

5.1. Distributed Processing

At the heart of our efficient processing design is the aggressive parallelization of analysis tasks. This parallel tasking occurs at several layers of the analysis cycle. The first such tasking is the analysis of each individual application. Each application can go through the processing and data extraction phases of analysis independently of each other, which incidentally is the most time intensive part of the process. We have designed the framework to take advantage of this fact and process each application in parallel rather than serially. In smaller implementations this is achieved using multi-threading, however at larger scales, we have implemented this using elastic computing and dynamically instantiated processing nodes, which allows an order of magnitude increase in processing speed.

We can then take this parallel processing a step further, by performing individual processing and analysis tasks on each application concurrently. Because many of the analysis tasks only require read-only access to resources such as the manifest, byte code, interfaces, or Smali code, it’s possible to complete each analysis task to extract the required information in parallel. Again, this distribution of tasking can be implemented using multi-threading or distributed nodes, though multi-threading on a single node reduces the amount of intermediate data that needs to be distributed. By using aggressive task concurrency throughout the analysis process, we are able to significantly reduce the latency for processing new incoming samples, and this approach also allows us to quickly adjust our analysis process to incorporate new analysis modules.

Task-oriented modules can be registered and unregistered with STAAF dynamically for each test. For instance, we’ve implemented high level modules to extract requested permissions from the manifest, extract statically defined URLs in the code, hash both Smali and Dalvik code sections, and extract a control flow graph of the Dalvik bytecode. Each of these modules can be turned on or off for a particular test, and each share underlying APIs to minimize the amount of processing and development time. For example, since multiple modules use the Smali version of the code, only the first module will need to convert the Dalvik bytecode into Smali, and all subsequent modules will only need to reference the original copy. By designing each module in a task oriented and compositional manner (e.g. extract dex file, convert dex to smali, extract all URIs, extract HTTPS URIs) we allow tasks to be run in parallel, while also benefiting from intermediate data produced by other modules.

5.2. Deduplication

Another core component that our framework uses to achieve efficiencies at scale is data deduplication. Data deduplication addresses two essential efficiencies, data storage and data processing. The most obvious problem of large-scale analysis is the explosion of data, which leads to a large amount of data to store and process. During our initial analysis using a previous version of our framework, we noted that many parts of the application, specifically libraries, were very similar if not the same. As these duplicate files and processed results began to aggregate, they consumed large quantities of disk space, and negatively impacted the performance of our database. Additionally, by processing the same code multiple times independently, a significant amount of time was spent duplicating efforts that had already been completed. We have addressed this in our new analysis framework, by storing intermediate results during each stage of the analysis process, and creating pointers in the database to intermediate and final results which analysis modules can reuse. By reducing the amount of duplicated processing that’s required our framework achieves both a speed increase and a reduced storage requirement.
5.3 Data Correlation and Analysis

Once each Android application is processed, the resultant data is stored in a distributed database, which can be efficiently updated and queried. From this database, an analyst has the capability to make complex queries and perform high level trend analysis for the entire collection, or a subset of the samples. For instance, in our initial implementation of the framework, we extracted the requested permissions, the method calls, the libraries used, and the URLs accessed for each application. This produced over 60GB of extracted data, which lead to a serious degradation in performance. To reduce this problem, we redesigned the framework to distribute the records across a set of distributed noSQL databases, which can be queried using map-reduce style queries. Because this map-reduce style query is designed for distributed database instances, this design also makes it easy to include remote third-party databases. By allowing teams and analysts to share processed results rather than raw data samples, each group can eliminate duplication of efforts simply by checking the data stores for the processed data.

These distributed databases can then be queried to determine high level trends, such as identifying any potentially malicious applications or traits using heuristics. In our initial version of this framework, we calculated the top requested Android permissions, the top shared code libraries, and the top contacted external URLs for a set of 50,000 applications. These seemingly simple task would take hours if not days to run, and in some cases would fail completely. By distributing the database and improving the query performance, we will be able to carry out much more complex data analysis tasks in a much more reasonable time.

6. Results

To test the performance and scalability improvements of the re-designed STAAF framework, we performed a series of nine tests with varying configurations as shown in the table below. In all tests we analyzed each application with modules that extracted: permissions declared in the manifest, static URIs in the Smali code, .invoke* calls in the Smali code, and a hash of each Smali code file.

<table>
<thead>
<tr>
<th>#</th>
<th>Time</th>
<th>Apps</th>
<th>ECU</th>
<th>Nodes</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2h25m</td>
<td>500</td>
<td>1</td>
<td>1</td>
<td>Central</td>
</tr>
<tr>
<td>2</td>
<td>2h00m</td>
<td>500</td>
<td>1</td>
<td>2</td>
<td>Central</td>
</tr>
</tbody>
</table>

Table 1. STAAF performance tests

In the table above we have listed the time that was required to process each set of applications. We initially started with a set of 500 applications to attempt several sub-optimal configurations for comparison, then performed two large scale analysis tests with optimal configurations. Note that these tests took place on Amazon AWS infrastructure and used both small instances (1 ECU) and medium instances (5 ECUs) for each node. According to Amazon, “One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.” Each node contained 1.7GB of RAM and 160GB of hard disk space.

Observing the test times one can see that central database storage, where all nodes log to the same database, prevents the distributed nodes from achieving optimal performance. Note that tests with the same number of processor nodes, that log to a local database rather than a central database, achieve 3x-4x speed improvements over those that log centrally. This is reinforced by comparing the CPU loads, where a database node remains at 100% throughout a test, and processor nodes without a database only utilize 20% of their available CPU. As one would expect, since this system appears to be bound by CPU, modifying each node to contain 5ECU CPUs achieves approximately 4x-5x improvement over 1ECU nodes. This fact actually allows us to decrease the overall cost of deploying STAAF since the cost of running a 5ECU instance is only twice the cost with four to five times the performance increase, allowing us to reduce the overall lifetime of the processing nodes and cut cost in half.

Finally in our large scale tests we show that STAAF does in fact successfully scale at an approximately linear rate. We sampled 1722 applications from the SlideME third-party marketplace (slideme.org), and were able to complete all processing in 27 minutes. The average processing time per application in test #7 was 4.8 compute seconds (600s*4nodes/500apps) and in test #9 it was 4.7 compute seconds (1620s*5nodes/1722apps). We then sampled 9349 applications from the official Android marketplace for a final test and achieved 5.1
compute seconds per application
\((4740s*10\text{nodes}/9349\text{apps})\). We believe that not only
can STAAF be scaled nearly linearly to larger
numbers of applications, but that the overall compute
time per applications can be further reduced.

7. Future Work

While we have achieved order of magnitude
increases in performance and scalability of the static
analysis of Android applications, we have identified
several areas of opportunity for future improvements.
First we have identified the noSQL database as a
significant bottleneck, and we believe this is due to
our design rather than an inherent limitation of the
database. STAAF was originally designed to log all
results to a single document, and using the document
model in CouchDB, continued to log all results for a
single application to a single document. In the
current design, these documents are written to
incrementally, as you would write to a local file,
however in CouchDB every write to a file requires a
full read of the file. Since some files can reach sizes
of hundreds of megabytes, this continuous stress of
reading the entire file to make a single update causes
unnecessary slowdowns in overall database
performance. We believe that addressing this by
breaking the analysis results into smaller individual
documents will eliminate this overhead and yield
additional performance benefits.

We would also like to augment future versions of
STAAF with additional task-oriented modules to
include many of the recent tools released and to
perform more complex analysis tasks such as static
data flow analysis and targeted source code analysis.
We believe that achieving performance and scale
efficiencies was a critical first step, and will allow us
to build more powerful analysis tools on top of the
STAAF framework.

8. Conclusion

Since the release of Enck et al. [27] and their
initial discussion about Android’s security features
and systems, researchers have been actively engaged
and contributing in various ways. With our work, we
are addressing several issues to help Android
application marketplace stakeholders deliver a safer
experience for their users and helping incident
responders more adequately address the challenge of
analyzing a large number of Android applications at a
single time. Our research began from a crude design
with no concept of runtime or static efficiencies, but
after observing the weaknesses in our initial
approach, we identified several fundamental changes
that were necessary to create a more robust and
efficient analysis framework. The product of our
research is a framework that scales efficiently,
enables analyst collaboration, and reduces the
duplication of effort and data.

The community at large has seen improved
awareness about potential attack vectors [1, 3, 4, 5]
and studies that have helped improve security and
privacy through applications and extensions [11, 21,
22, 23, 24, 25, 28], however, there has been very
little movement in the direction of improving the
active prevention of malware in Android markets,
and there appears to be even less movement in
improving the analysis of applications during the
course of an incident response case. To address the
needs and challenges described, we have proposed
STAAF. Our framework emphasizes distributed and
parallel data processing, task-oriented modules, and
enables data sharing and high level trend analysis.
We hope that the release of this framework will
encourage the proactive analysis of Android
applications, and identify potentially malicious
applications before they’re able to infect a user’s
device.

9. References

[1] Schlegel, R., Zhang, K., Zhou, X., Intwala, M.,
Kapadia, A., and Wang, X., "Soundcomber: A Stealthy and
Context-Aware Sound Trojan for Smartphones" Proc. of
the 18th Annual Network and Distributed System Security
Symposium (NDSS), NDSS, San Diego, CA, Feb. 2011, pp
17–33.

[2] Xiang, C., Binxing, F., Lihua, L., Xiaoyi, L., and
Tianing, Z., "Andbot: Towards Advanced Mobile
Botnets", Proc. of the 4th USENIX conference on Large-
scale exploits and emergent threats (LEET), USENIX,
Boston, MA, Mar. 2011.

and Winandy, M. "Privilege Escalation Attacks on
Conference (ISC), Springer, Boca Raton, FL, Oct. 2010, pp
346-360.

"Analyzing Inter-Application Communication in Android",
Proc. of the 9th Annual Int’l Conf. on Mobile Systems
(MobiSys’11), Applications, and Services, ACM,

[5] Shin, W., Kwak, S., Kiyomoto, S., Fukushima, K., and
Tanaka, T., "A Small but Non-negligible Flaw in Android


[28] Zhou, T., Zhang, X., Jiang, X., and Freeh, V., "Taming Information-Stealing Smartphone Applications (on
