Prepared for *CS263: Systems Security*, Harvard School of Engineering and Applied Sciences, Prof. J. Mickens.

SimpleTimingPwn: Evading Information Flow Analysis via an Extremely Simple Timing Channel

Dan Fu Ross Rheinans-Yoo

Harvard University

dfu@college.harvard.edu rry@eecs.harvard.edu

Abstract

Confidentiality of sensitive information is crucial to the security of modern mobile phone operating systems. To address this concern, security researchers have proposed varied methods for detecting exfiltration of sensitive data via both 'dynamic' and 'static' information flow analysis, often acknowledging their limitations in detecting side-channel communication, such as timing-channel methods. However, we demonstrate that such a side-channel attack is extremely trivial to design, presenting a simple clock-based protocol for exfiltrating arbitrary strings of data using timing signals in 10 lines of Java code. We present an implementation of our protocol, demonstrate that it avoids detection by popular tools for both dynamic and static analysis, and that it is capable of cleaning data for exfiltration at rates of several bytes a second. While this is too slow for exfiltrating large quantities of sensitive information, we demonstrate that it is more than sufficient for exfiltrating IMEI codes, location data, other identifying tokens, and even alarming quantities of private contact information.

1. Introduction

68% of American consumers own a smartphone, compared to 73% who own a desktop or laptop computer [1], to say nothing of the enormous worldwide usage, passing 2.6 billion users [2]. Of those, more than 70% use Android [3], a Linux-based, open-source operating system whose functionality is extended by Google Play [4], a centralized service for downloading third-party applications which has served more than 50 billion downloads [5] of more than 1.8 million provided applications [6].

The Android OS, furthermore, is the target of 98% of malware applications targeting smartphones [7]. In contrast to the personal-computer paradigm, where most sensitive information is both difficult to locate and illegal to abuse, mobile phones are rife with easily-accessed, personally-identifiable or otherwise sensitive information of significant interest to advertising companies, either for their own use or for commercial resale. Access to this information is often controlled by coarse-grained permissions controls, but the ultimate *use* of any information permitted is effectively uncontrolled, and a 2010 survey of the Google Play market revealed that approximately one third of the most popular applications required access to both the Internet network interface and location, camera, or audio data, [8] let alone personally-identifiable information such as the (device-unique) IMEI code, or other sensitive information, such as contact data.

While a variety of methods have been developed to detect malicious information flows in traditional computer systems [9] [15] and web applications [16], Enck *et al.* identify four ways in which the problem of monitoring information flow is more difficult on smartphones than in traditionally-studied systems:

- "Smartphones are [relatively] resource constrained."
- "Third-party applications are entrusted with several types of privacy-sensitive information," which may require independent tracking.
- "Context-based privacy-sensitive information. . . can be difficult to identify even when sent in the clear."
- "Applications can share information."

These challenges, combined with novel sorts of sensitive information to discover and exfiltrate, necessitate a renewed and redesigned approach to information flow analysis.

Fortunately, the security community has not been completely negligent in this regard. We discuss published technologies for information flow analysis (in section 2), discuss a basic side-channel protocol which exfiltrates information entirely undetected by these methods (in section 3), describe our implementation and testing of our exploit (in section

4), discuss its practical limitations and security implications (in section 5), and conclude with general observations and lessons for mobile information security (in section 6).

2. Background: Information Flow Analysis

2.1 Dynamic Analysis: TaintDroid

One versatile tool for detecting application-level leaks of sensitive information at runtime is *dynamic taint analysis*, which assigns 'taints' to sources of sensitive information, and thereafter propagates them across variable assignments, calculation results, interprocess messages, *etc*. Then any attempt by an application to pass a tainted variable to an external network interface or other communication channel can be interpreted as a (potential) attempt to exfiltrate some derivative of the original sensitive information.

TaintDroid, designed by Enck *et al.* is one implementation of this technique for the Android platform [7] which tracks taints in real-time during program execution by:

- instrumenting Android's JVM interpreter to track variable-level assignments within a process,
- instrumenting the Android system's inter-application message passing logic to track data passed between processes,
- patching native libraries to track taints at the method level to track data mutated by standard library calls,
- and patching system-level filesystem logic to track data written to persistent storage in files.

In 2010, the authors used TaintDroid to examine 30 popular Android applications, and "found 68 instances of potential misuse of users' private information across 20 applications." In a longitudinal follow-up study conducted in 2012, they found that 12 of the 18 applications for which updated versions could be obtained had persistent or new instances of misuse [8].

Their work extended that of previous authors who had applied dynamic analysis to find information leaks in a desktop setting, either by environment emulation [9] [10], library instrumentation [11] [12], or JVM hooks [13], updating their work to tailor it to the demands of the smartphone computing environment and the specific features of known data sources.

Previous work in information flow analysis in smartphones presented the possibility of intercepting outgoing messages which contained certain known sensitive substrings [], though, as Enck *et al.* note in the original TaintDroid paper [7], this is easily circumvented by even the most trivial encryption schemes.

2.2 Static Analysis

However, dynamic analysis has functional limitations which prevent it from detecting certain data-leakage channels, including *implicit control flows*, whereby control flow logic can leak a few bits of information at a time while avoiding direct value-assignments which would cause taint propagation [17]. Graa *et al.* present a simple example[1]:

```
1:  function IMPLICITCLEAN(a)
2:      b ← false
3:      c ← false
4:      if not a then
5:          c ← true
6:      end if
7:      if not c then
8:          b ← true
9:      end if
10:     return not b
11: end function
```

Note that if a is true, then line 5 is not executed (and so c is not tainted), but the b is set to be true on line 7. If a is false, then line 5 *is* executed, but line 8 is not, and so b is left as false. In neither case is b tainted as of its return on line 10.

While understanding the relationship between the values of variables in control-flow statements and the code executed in the controlled logic is necessary to detect this sort of leak, it is important to realize that it is not enough to propagate taints from the variables which control, say, an **if** to any variables set inside the relevant block. In Graa *et al.*'s example above, for example, such an approach would still fail to properly propagate a's taint to b.

Instead, any analysis approach must take into account the logic which may be counterfactually executed, and the logic invoked for *any possible value* of variables conditioned upon. Such methods are known as *static analysis*, since investigating the counterfactual codepaths is performed statically, rather than through (dynamic) real-time observation of a particular run.

Graa *et al.* provide both a theoretical model [17] and a working implementation [18] of an analysis engine which uses static analysis to determine implicit information flows, then dynamic analysis to monitor (and prevent) disallowed taint-to-sink propagation. Their work builds on prior work in static analysis of known conditional idioms [19], the application of these techniques to inform runtime dynamic analysis in a traditional computing setting [20], and analysis of certain implicit flow idioms based on Fenton's Data Mark Machine model [21]. In particular, they use the more-general information-lattice model proposed by Denning [22] to reason about implicit flows, then apply the combined static/dynamic approach of BitBlaze [20] within the TaintDroid instrumentation framework [7].

[1] Graa, Cuppens-Boulahia, Cuppens, and Cavalli present this example in a paper titled "Detecting control flow in Smarphones" *[sic]*, which was accepted to the 4th IEEE International Conference on Cyberspace Safety and Security, despite their failure to explain anywhere in their paper what exactly a 'smarphone' is. This realization served to further undermine our already-thin faith in the power of academic peer review.

3. Basic Clock-Based Communication

3.1 Algorithm and Explanation

We present an algorithm for transfering relatively short strings of tainted data to untainted variables which evades detection by either dynamic or static analysis, as follows:

```
1:  function TIMINGCLEAN(x, ℓ, t)
        ▷ x is a ℓ-long bitstring; t is the timing threshold.
2:      r ← ALLOCBITS(ℓ)
3:      for i ∈ [0, ℓ) do
4:          pretime ← SYSTEMTIME
5:          if x[i] then
6:              SLEEP(2t)
7:          end if
8:          curtime ← SYSTEMTIME
9:          if (curtime − pretime) > t then
10:             r[i] ← 1
11:         else
12:             r[i] ← 0
13:         end if
14:     end for
15:     return r
16: end function
```

The contents of the functional loop are lines 4 through 13, inclusive. This logic leaks a single bit of information via the system clock—or more precisely, via the difference between the current value of the system clock and the recorded value. If the relevant bit of the sensitive data was set, the difference will be large (in particular, larger than the threshold t), but if the bit is not set, then the difference will be negligible (and thus much less than t, for appropriate values of t).

3.2 Analysis and Potential Methods of Detection

Assuming that SYSTEMTIME is not tainted and SLEEP has no side effects and touches no taint sinks, this algorithm does not propagate taint from x to r under either dynamic or static analysis. Under dynamic analysis, x's taint is propagated nowhere; under static analysis, it is propagated to the call to SLEEP, but since the latter has no side effects, no further. In either case, r will carry no taints, and can be sent across the network without detection.

While Graa *et al.* attempt to prove formally that static analysis avoids under-tainting such as this, the mere introduction of a system clock violates the Denning information-lattice model [22] upon which they rely. More troublingly, a large variety of external data sources can serve in this role, since the 'clock' need only be able to distinguish between near-instantaneous and aribitrarily-long time delays. Hence, attempts to taint SYSTEMTIME itself can be avoided by modifying the algorithm to use battery charge level[2], device temperature[3], or other innocuous and untainted sources which vary over time. Alternatively, a parallel timer application (itself using SLEEPs or some expensive computation to keep rough time) or even Internet access to a third-party web server (potentially controlled by the exfiltrator) can serve the same role, while being impossible to taint without causing a taint explosion.

Similarly, it is infeasible to mark SLEEP as a taint sink, given the significant potential for false positives, and in any case the ease of developing around it by invoking some expensive computation or other time-consuming method (such as file I/O, or even innocuous user prompts) instead.

4. Implementation, Testing, and Analysis

4.1 Android Implementation Details

To test our hypothesis that we can use timing channels to leak information while avoiding taint detection, we wrote three apps that sent the IMEI code of the device to a remote server in three different ways. The IMEI code is a fifteen digit code unique to every mobile device and is indicative of the type of data that taint propagation aims to protect - there is only one way for an app to get the information (a system call), so it is easy to introduce a taint at that point of contact. Once the code has been tainted, the taint can be propagated throughout the code's lifetime, and TaintDroid should be able to detect when the code is sent to a remote server.

Our three apps are functionally identical except for the ways that they internally process the IMEI code once they receive a tainted version from the Android operating system. Each app has a button creatively titled "Leak my data" that, when pressed, calls a service to get a tainted IMEI code, processes it, and sends out the IMEI code to a remote server. For our purposes, we simply send out the IMEI code to a free HTTP Post dumping server [23] and manually verify that the received IMEI code is correct.

We use our first app, a naïve data leaker, as a control case to verify that TaintDroid indeed works as advertised. The naïve data leaker does not attempt to evade the taint in any way - it simply sends the same piece of memory that is returned by the system call to get the IMEI code.

Our second app uses an implicit control flow to avoid taint detection. We loop through each digit of the tainted IMEI code and compare the digit to each of $0 - 9$. If we find a match, we add the matched digit to an untainted IMEI code that we are constructing. In this way, we construct an untainted piece of memory that has the same data as the original tainted IMEI code. Then, we simply send out the untainted IMEI code through the network instead of using the tainted IMEI code.

[2] The third-party Dropbox application, for example, avoids battery-intensive syncing operations when the battery is below a certain level of charge, and is an example of the sort of application which would appear as a false positive if battery information is tainted.

[3] While device temperature doesn't evolve monotonically over time as do time or charge, we can easily modify the algorithm to replace the SLEEP with a loop that waits until it changes more than a certain amount away from its recorded base value, then breaks.

Finally, our third app uses timing channels to avoid taint detection. Like our second app, we loop through each digit of the tainted IMEI code to construct a new piece of memory that has the same information as the original tainted IMEI code but which is untainted. For each digit, we record the current system time, read the value d of the digit, and sleep for $100 + 100d$ milliseconds[4]. After waking up, we subtract the current system time from the recorded system time and use it to guess what the original digit from the tainted IMEI code was. We add this digit to an untainted piece of memory. After looping through all the digits of the original IMEI code, we send out the untainted IMEI code through the network. To avoid the problem of our timing attack bringing the UI to a halt and triggering Android's warnings about unresponsive apps, we simply run the timing attack in a background thread, which Android makes easy by providing Java's simple interface for running background threads.

We would like to emphasize how little effort is required to transition from a naïve data leaker to one that uses timing channels. In particular, once we had the naïve data leaker running, it only took roughly 10 lines of code to turn it into a data leaker that used timing channels. To give some context, while it took 10+ hours to successfully install TaintDroid and create a testing environment, it took roughly 3 hours to develop the three apps that leaked the IMEI code in different ways.

4.2 TaintDroid Analysis

We installed our three data leakers on a system running TaintDroid and had each one send the IMEI code to a remote server. As expected, TaintDroid successfully caught the naïve data leaker, but it did not catch either the implicit control flow leaker or the timing channel leaker. Furthermore, while we observed a slight delay between the times when we clicked the "leak my data" button and when the server received the timing-channel-cleaned IMEI code, the delay was never more than 3 seconds long. With the leak happening in the background, there was also no noticeable UI lag.

4.3 Static Analysis

We would like to have run our three apps on a system with the static-analysis infrastructure proposed by Graa *et al.* to avoid under-tainting, but the authors have not released the source code for their analysis engine. Instead, we can formally demonstrate that Graa *et al.*'s own proof model [17] fails to recognize the timing signal in TIMINGCLEAN as an implicit flow.

Their static-analysis proof model understands flow of information from variables to conditions, conditions to variables, and variables to variables. So, stepping through the TIMINGCLEAN pseudocode above, we see the following taint propagations:

2. $r \leftarrow \text{ALLOCBITS}(\ell) \Rightarrow \text{Taint}(r) \leftarrow \text{Taint}(\ell)$
3. $\text{pretime} \leftarrow \text{SYSTEMTIME} \Rightarrow$ $\text{Taint}(\text{pretime}) \leftarrow \text{Taint}(\text{SYSTEMTIME})$
4. **for** $i \in [0, \ell)$ **do**...**end for** $\Rightarrow \text{Taint}(i) \leftarrow \text{Taint}(\ell)$
5. **if** $x[i]$ **then** $\text{SLEEP}(2t)$ **end if** $\Rightarrow$ $\text{Taint}(\text{SLEEP}) \leftarrow \text{Taint}(x) \oplus \text{Taint}(t)$
8. $\text{curtime} \leftarrow \text{SYSTEMTIME} \Rightarrow$ $\text{Taint}(\text{curtime}) \leftarrow \text{Taint}(\text{SYSTEMTIME})$
9. **if** $(\text{curtime} - \text{pretime}) > t$ **then** $r[i] \leftarrow 1$ **else** $r[i] \leftarrow 0$ **end if** $\Rightarrow$ $\text{Taint}(r) \leftarrow \text{Taint}(\text{pretime}) \oplus \text{Taint}(\text{curtime}) \oplus \text{Taint}(t) \oplus \text{Taint}(i) = \text{Taint}(\text{SYSTEMTIME}) \oplus \text{Taint}(t) \oplus \text{Taint}(\ell)$

Note that the taint on x is propagated nowhere but to the call to SLEEP (which, as a system-lvel call, propagates taints to its return values and side-effects—of which there are none), while r has adopted the taints of SYSTEMTIME (which, as we argued in section 3.2, cannot feasibly be tainted), t, and ℓ. Since t is a constant corresponding to the timing sensitivity and ℓ is the length of the data (presumably known ahead of time, for information such as the IMEI or location readings), neither need be tainted since both can simply be hardcoded. So r will carry no taints from x, and indeed, no taints at all.

The decimal-encoding approach we employed in our implementation has identical taint assignments, and by similar proof successfully copies the information into a clean variable for return.

5. Implications and Limitations

5.1 Limitations of Timing Channels

Our implementation was successful in a laboratory setting, though timing channels may be expected to have limitations in practical settings.

First, the timing attack takes much longer than the implicit control flow attack, owing to the usage of explicit-duration sleeps—potentially up to 15 seconds for a 15-digit IMEI code, though that time could be reduced with a smaller sleep time and more efficient algorithm. Our choice of 100 milliseconds per timestep was rather arbitrary, and could be tuned for a particular usage-environment (even dynamically, by a clever application), though we expect that scheduler nondeterminism will place a lower bound on how short we can make relevant time-interval steps. A more-efficient encoding scheme (binary, rather than decimal, as in our case) could also serve to decrease runtime, though not by more than a multiplicative factor. While this extra time is spent sleeping rather than computing (and so plays somewhat nicely with concurrent applications), it nevertheless

[4] This is a slight modification from the TIMINGCLEAN algorithm presented above, intended to simplify the encoding/reconstruction of decimal information. A time-signal length of 100 milliseconds per timestep was roughly determined to be adequate for communication, but could easily be lowered (to speed up cleaning potentially at the cost of accuracy) or raised (to increase reliability at the cost of speed).

places practical limitations on the rate at which data can be exfiltrated.

The second problem is that explicitly making a thread sleep introduces the risk of the thread being de-scheduled for much longer than anticipated, thus throwing off the reconstruction of the sensitive data. At best, we get a clearly-invalid time-interval and can re-attempt, but it's possible that we accidentally get an incorrect interval which appears to be valid. We did not experience this problem in the testing environment, but in a real setting, users may very well be running multiple apps at the same time. While we can eliminate potentially-errant sleeps by, again, using a binary encoding scheme, rather than our decimal encoding, we are nevertheless bound to using some sort of system call to observe our clock time (or status, for alternative clock-like channels), which will by design have to allow for system blocks and descheduling.

One solution to this is to take multiple measurements of each digit and take the minimum of the measurements. This would increase the overall time it takes to reconstruct the IMEI code, but would help sample out scheduler effects, since it requires only that a single run (of arbitrarily many) be re-scheduled within less than the time threshold of the requested sleep time.

Another solution would be to use expensive computations to introduce non-blocking time delays, though proper implementation would need to be carefully adjusted to scheduler dynamics in order to avoid inadvertent scheduler preemptions. We do not present implementations of these solutions, however, as our timing algorithm was, in practice, sufficient.

5.2 Security Implications

Both the original and follow-up TaintDroid papers conclude with the sentence "Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with TaintDroid." Graa *et al.*'s original static-analysis paper concludes:

> We prove that our system cannot create under tainting states. Thus, malicious applications cannot bypass the Android system and get privacy sensitive information through control flows... Once the implementation is finished, we will be able to evaluate our approach in terms of overhead and false alarms. We will also demonstrate the completeness of the propagation rules.

The authors' follow-up paper, describing said finished implementation, concludes that " By implementing our approach in Android systems, we successfully protect sensitive information and detect most types of software exploits caused by control flows." [18]

Notwithstanding these claims, we demonstrate that roughly 10 lines of Java code evade these methods of control-flow analysis in a matter of seconds, and are more than capable of leaking IMEI codes, location data, and other short strings of sensitive information to external servers. Since our timing-based cleaning is capable of running in the background without noticeable UI lag, even our simple timing-channel attack is capable in theory of processing several kilobytes an hour, sufficient to exfiltrate, for example, about a hundred contact name/number/email entries, if properly compressed, in that time.

Moreover, since our timing channel can be applied to arbitrary computed data, it can be employed at any step in a data-exfiltration pipeline. A malicious application can perform any arbitary precomputation or compression before 'cleaning' the resulting (tainted) data, and any desired encryption to evade detection [14] before sending the information in the clear.

For example, while high-bandwidth data from camera and audio sensors may not be so easily exfiltrated wholesale, a malicious application eavesdropping on phone audio can recognize audio cues from either people or nearby audible advertisements [24], and report over the network a clean message that such a cue was detected. Similarly, continuous accelerometer data sufficient to monitor a user's physical behavior [25] can be parsed to produce discrete information terse enough to be cleaned and transmitted across the network in close-to-realtime.

6. Conclusion

We present a timing-channel exfiltration algorithm that evades detection by both dynamic and static analysis. We reiterate that, despite sounding like l33t h4x0r witchcraft, this is by no means a sophisticated attack—in fact, it was designed by two undergraduates with nothing better to do, and took less time to develop and test than it did to install and set up an Android build environment.

Our experiences have supported the pessimistic conclusion that if you do not wish for all of your base to belong to advertisers (and other h4x0rs), information-flow analysis methods are insufficient to protect sensitive information on mobile phones. Instead, with the possible exception of high-bandwidth data (*e.g.* from microphone and camera sensors—though not feature-extracted or semantic data derived therefrom), it is likely necessary to prevent untrusted third-party applications from any access to information that should not be leaked wholesale over the network. Nevertheless, surveys of the Android application market have found that a third or more of third-party applications require access to both sensitive information and the Internet.

We conclude that mobile information security is pwned.

Acknowledgments

The authors would like to acknowledge the contribution of James Mickens of Harvard, whose course (and, perhaps most directly, its requirement of a final project) motivated and informed much of our research. Various "... for Dummies" tu-

torials on the Internet were invaluable in coaching us through the process of setting up an Android build/simulation environment, which, as previously noted, was significantly more difficult than finding major holes in previously-published and peer-reviewed security research.

References

[1] M. Anderson. Technology device ownership: 2015. *Pew Research Center*, October 2015. `http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015/` ret. 29 Nov 2015.

[2] A. Boxall. The number of smartphone users in the world is expected to reach a giant 6.1 billion by 2020. *Digital Trends*, June 2015. `http://www.digitaltrends.com/mobile/smartphone-users-number-6-1-billion-by-2020/` ret. 29 Nov 2015.

[3] L. Whitney. Android market share stays steady in US but sinks deeper in Europe. *CNet*, September 2015. `http://www.cnet.com/news/android-market-share-stays-steady-in-us-but-sinks-deeper-in-europe/` ret 29 Nov 2015.

[4] Google Play. `https://play.google.com/store` ret. 29 Nov 2015.

[5] C. Warren. Google Play hits 1 million apps. *Mashable*, July 2013. `http://mashable.com/2013/07/24/google-play-1-million/` ret. 29 Nov 2015.

[6] AppBrain. Number of Android applications. November 2015. `http://www.appbrain.com/stats/number-of-android-apps` ret. 29 Nov 2015.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. OSDI'10*, 2010.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 32(2):no. 5, June 2014.

[9] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. CCS'07*, 2007.

[10] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proc. ISSTA'07*, pages 196–206, 2007.

[11] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. MICRO 39*, pages 135–148, 2006.

[12] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy Scope: A precise information flow tracking system for finding application leaks. Tech Rep. EECS-2009-145, Dept. Comp. Sci., UC Berkeley, October 2009.

[13] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. *Elec. Notes in Theo. Comp. Sci.* 197(1), pages 3–16, February 2008.

[14] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *Proc. NSDI'07*, 2007.

[15] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieères. Making information flow explicit in HiStar. In *Proc. OSDI'06*, 2006.

[16] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proc. SOSP'09*, October 2009.

[17] M. Graa, N. Cuppens-Boulahia, F. Cuppens, A. Cavalli. Detecting control flow in smarphones: combining static and dynamic analyses. In *Proc. CSS'12*, pages 33–47, 2012.

[18] M. Graa, N. Cuppens-Boulahia, F. Cuppens, A. Cavalli. Detection of illegal control flow in android system: protecting private data used by smartphone apps. In *Foundations and Practice of Security*, Lecture Notes in Computer Science 8930, pages 337–346, April 2015.

[19] M. G. Kang, S. McCamant, P. Poosankam, D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. NDSSS 18*, 2011.

[20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena. Bitblaze: A new approach to computer security via binary analysis. *Info. Sys. Sec.*, pages 1–25, 2008.

[21] J. Fenton. Information protection systems. Ph.D. Thesis, U. Cambridge, 1973.

[22] D. Denning. A lattice model of secure information flow. *Comm. ACM* 19(5), pages 236–243, 1976.

[23] Henry's HTTP Post Dumping Server. `https://www.posttestserver.com/` ret. 28 Nov 2015.

[24] B. Schneier. Ads surreptitiously using sound to communicate across devices. *Schneier on Security*, November 2015. `https://www.schneier.com/blog/archives/2015/11/ads_surreptitio.html` ret. 29 Nov 2015.

[25] M. Fitzpatrick. Mobile that allows bosses to snoop on staff developed. *BBC News*, March 2010. `http://news.bbc.co.uk/2/hi/technology/8559683.stm`