Practical Always-On Taint Tracking on Mobile Devices

Justin Paupore[†], Earlence Fernandes[†], Atul Prakash[†], Sankardas Roy[‡], and Xinming Ou[‡]

[†]University of Michigan, Ann Arbor ({jpaupore,earlence,aprakash}@umich.edu)
[‡]Kansas State University ({sroy,xou}@ksu.edu)

Abstract

Taint tracking is a crucial yet expensive security primitive. In the context of mobile devices, given the volume of sensitive data being generated and manipulated, taint tracking is an important aspect of defense in depth, yet is not widely adopted due to performance and energy constraints. Existing work has proposed several forms of optimization for desktop based systems – software-only mechanisms, static analysis, hybrid analysis, and hardware-assisted techniques. This paper makes the case for an always-on taint tracking system for mobile devices that embraces the unique properties of mobile operating systems – interpreted runtimes, well-defined APIs, and an overlooked ARM processor feature. Our proposed system combines precise static analysis on Java code and real-time instruction trace support widely available on ARM processors to enable efficient taint tracking.

1 Introduction

Modern smartphones carry a wealth of data. The combination of always-available Internet access and a plethora of sensors allow app developers to create rich applications that would be difficult, or even impossible, to create on other platforms. However, that capability comes with a downside - that same combination of Internet access, personal information, and sensor data make smartphones a goldmine for identity thieves, stalkers, and other malicious actors.

To mitigate this risk, smartphone operating systems include mechanisms for controlling apps' use of data. The mechanism used by major smartphone OSs today involves a system of permissions - apps must explicitly request, either at installation time or at runtime, access to sensitive information or communications services. While this model is easy to understand and straightforward to implement, it has several weaknesses in practice. Chief among those weaknesses is the issue of dangerous permission combinations - once an app has access to private data and a way to get it off the device, it can, whether by accident or by intent, leak that private data without the user's knowledge.

The canonical solution to the information-disclosure problem is to use *taint tracking*. Taint tracking keeps track of private data as it moves through a program's code, from a sensitive source such as GPS, through to an off-device sink such as the Internet. Any computation that depends on tainted data itself becomes tainted, and if any tainted data reaches a sink, a leak of private data is detected. The system can then take action, potentially including blocking the leak and/or replacing the tainted data with garbage.

Taint tracking is a powerful tool, but it is often a very expensive one. Even state-of-the-art, optimized taint-tracking systems for binary code typically cause 2.75x slowdowns in the code being tracked [8]. Systems that operate on more restricted input, e.g., TaintDroid [7] for DEX bytecode from Java, have better performance, but still impose a worst-case overhead of 30%.

An alternative approach to dynamic taint tracking, as in the above systems, is to do static code analysis to identify data flows that may leak private information. FlowDroid and Amandroid [3, 23] are examples of such tools. Static analyzers have the advantage of zero runtime overhead, along with the ability to perform analyses across the program, without being restricted to the paths actually taken at runtime. However, without the program's inputs, static analyzers can't tell how often a given flow occurs in practice, or even whether it happens at all. This is an important distinction to make for smartphone apps - an app that sends your location to a server when the user asks for nearby restaurants, for example, is much different in practice from an app that does so constantly in the background.

The best possible system, therefore, would be a *hybrid* system combining the runtime knowledge of dynamic taint tracking with the full-program knowledge of static

analysis. Such systems have been discussed in the past, both in theory [11, 10, 9] and in practice [8, 12]. However, existing work on this subject has focused on desktop and server workloads, with very little work focusing on the mobile environment.

The remainder of this work proceeds as follows: Section 2 describes the constraints and features of smartphones. Section 3 proposes a system for efficient taint tracking within that framework. Finally, Section 4 discusses related work and concludes.

2 Design Space

Smartphones, while similar in many respects to desktop systems, have some key differences that make their software environment quite different. First and foremost, smartphones are limited in computational power. Most current desktop machines have enough resources on tap that even heavily CPU-bound workloads tend to complete quickly. In contrast, smartphones have much slower and less capable CPUs, and rely on hardware acceleration to handle common CPU-intensive tasks such as media encoding. This constraint means that most static analysis methods, which tend to be very CPU-intensive, cannot be performed on-device. It also makes even small overheads in execution time more noticeable to the end user.

Additionally, smartphones are energy-constrained. Smartphone users demand the ability to go a full day without a charge, causing hardware and operating systems vendors to go to great lengths to minimize the amount of energy used. To save energy, smartphone processors are designed to shut down peripherals and cores that are not in use, and their software is designed to batch requests together to minimize the amount of time those peripherals must stay on. This requirement rules out any cloud-based computation that requires continuous network access, and imposes additional costs on CPU time used by dynamic analysis.

Fortunately, smartphones also have a few features that simplify the problem of taint tracking, even under those constraints. First of all, both Android and Windows Phone apps run under a managed, typesafe runtime. The semantics of that runtime allow the system to make strong assumptions about how code can behave, which simplifies the logic required for taint tracking. Additionally, these runtimes use ahead-of-time compilation, which allows us to control the machine code executed for a given bytecode instruction without needing hard-to-predict branches inside an interpreter. (See section 3.1.)

Second, every smartphone OS has an app store, which users trust to provide them apps. The app store operators have a vested interest in their users' security, and a large infrastructure at their disposal, which provides a perfect place to perform heavyweight static analysis, which can be used on-device to cut down the runtime overhead of taint tracking or skip it entirely (see section 3.2).

Lastly, every popular smartphone on the market uses an ARM processor. These processors are highly-integrated pieces of hardware, containing a wide array of coprocessors, modems, and peripherals that are used to perform specialized tasks. In particular, every ARM chip shipped in the last several years has included a peripheral called an Embedded Trace Macrocell, or ETM. While ETM was designed to collect execution traces as a debugging aid, it can be repurposed to send execution information from an app to another core, enabling the app's execution and taint propagation to happen in parallel (see section 3.3).

3 Proposed Design

In our proposed design, the analysis begins when an app is uploaded to the app store by its developer. The app store operator runs a static analyzer over the app, and saves the resulting output along with the app binary. When a user installs the app, the app store also sends the static analysis results. The app and the static analysis results are sent to the ahead-of-time compiler, which generates ARM machine code for the app. As it generates the code, it emits marker instructions into the instruction stream, guided by the static analyser output.

When the app is launched, the operating system configures the ETM trace unit for a core to generate an event when a marker instruction is executed. The app is then run on that core. Every time the app hits a marker instruction, the ETM trace unit generates an event containing the marker instruction's PC and an operand determined by the static analyzer output, and places the event in a trace buffer. At the same time, on another core, a taint-collector thread reads these trace events. By using the PCs and data sent from the main app, the taint collector thread reconstructs the app's stream of execution, and from there, can infer changes to the app's taint state. When it infers that a leak has occurred, it logs the leak and notifies the user.

The remainder of this section discusses the design in more detail. This work simply lays out the design; we leave it to future work to implement it and verify its functionality.

3.1 Semantic Analysis

Unlike traditional native-code-based desktop applications, mobile apps tend to be written in managed languages such as Java on Android and C# on Windows Phone. These languages provide full type- and memory-safety, cross-architecture support, and garbage col-

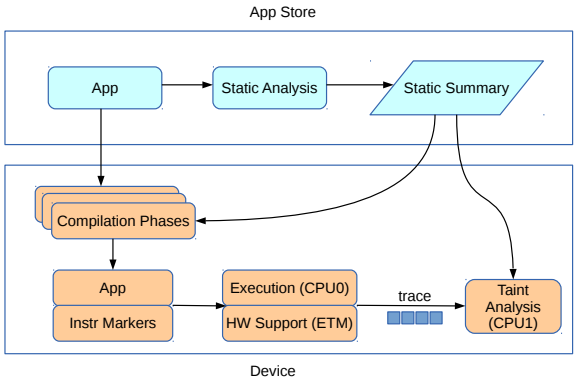

Figure 1: A diagram of our optimized taint tracking system. Static analysis performed on a central app store is combined with on-device dynamic analysis, which is decoupled using ARM's ETM trace unit.

lection as part of their design. In order to support those services, app code is distributed in an intermediate language. These intermediate languages carry a large amount of semantic metadata, including class and interface definitions, method bytecode, inheritance and override information, and debugging data. That data can be used in multiple ways to simplify the problem of taint tracking. For example:

- Managed bytecode is structured into clearly defined methods, classes, and interfaces. This structure allows unambiguous disassembly and reconstruction of the original control flow graph, enabling static analysis of apps without needing access to source code that app developers may be unwilling to provide. In contrast, native machine language is far more difficult to disassemble and analyze, with unambiguous x86 disassembly being a provably undecidable problem. [22]
- Bytecode is type- and pointer-safe, simplifying static analysis by reducing the possible aliases of a reference to other references of compatible type.
- Objects are clearly separated in managed languages, allowing taints to be tracked per-object and per-field, rather than per-byte.

This additional data allows for lower-overhead taint tracking for the vast majority of apps that use bytecode. For those few apps that take advantage of native machine code (for example, via Java's JNI or Android's NDK), a more traditional native-code approach can be used.

In addition to the direct benefits of using managed languages, mobile taint tracking also benefits from the properties of mobile OS API designs. By design, these APIs funnel all access to sensitive information through a small number of well-defined interfaces, where permissions can be checked. These interfaces serve as natural points to apply taint markers. Sinks follow a similar pattern, again due to the need for permissions checking.

The work done to enforce permissions also provides an extra benefit due to the association of permissions with sources and sinks - by taking advantage of the different permission types that are used, taints from different sources can be distinguished. For example, leaks of a user's location can be distinguished from those involving the user's contacts. Projects such as SuSi [16] have explored the feasibility of performing this analysis automatically, thus reducing the engineering work that is required to annotate those sources manually.

3.2 Cloud-Assisted Hybrid Analysis

As previously mentioned, the best possible taint tracking system would take advantage of both static knowledge about the program's structure and runtime knowledge of the program's behavior. However, implementing this hybrid analysis has several challenges. The first challenge is the issue of performing the static analysis in the first place. Most analyzers are very CPU- and RAM-intensive, making them poorly suited to run on low-power smartphone chipsets. The solution is to farm out the analysis to the operator of the device app store, who would perform the analysis on a much less resource-limited machine, then sign the analyzer results and include the signed results in the app download. This allows the analysis to be done once per app uploaded, then served to any number of customers who need it.

App store vendors are already performing some level of security checks on submitted apps and updates, either automatically as in the case of Android, or manually as in the case of iOS. Therefore, even a very long-running, computationally-intensive static analysis – such as that performed by TAJ [20], which can take over an hour to run a complete analysis on a large app – would not be a disruption to the current app-development ecosystem. This means that an expensive, but precise, static analysis can be run on each app, which would reduce the amount of work left to be done by the runtime taint-tracking system. Instructions that don't deal with private data, and that can be proven safe, can be skipped entirely in the runtime analysis. Previous work on published apps has shown that private data is generally only handled in a small percentage of code [24], so excluding the rest of the app from consideration should reduce the overhead considerably.

Another significant challenge involves handling taint from other apps. Android makes heavy use of IPC between processes to share information. That information may be tainted, but it won't always be. Therefore, each app needs to treat any data received via IPC as potentially tainted, and thus perform taint analysis of that data.

Similarly, data from the local filesystem may also be tainted. It remains to be seen what impact these additional sources and sinks will have on the effectiveness of the static optimizations.

The final challenge involves finding an efficient way to customize the runtime taint-propagation behavior. With a conventional interpreter like Android's Dalvik, which uses the same implementation code for every instance of a given bytecode instruction, this would require a conditional branch on a hot path with a difficult-to-predict pattern. Under those conditions, branch mispredicts may, in fact, make the implementation slower than simply propagating every time [5]. To avoid that issue, it would be necessary to have separate native implementations for each bytecode instruction; preliminary investigations into ART, Android's ahead-of-time-compiled Dalvik replacement, show that it is possible to modify the code generator to emit additional native instructions into an app binary based on an input file. This capability could be used to emit taint-propagation logic only on instructions that require it, saving clock cycles and memory accesses.

3.3 Hardware-Assisted Taint Collection

While using hybrid analysis can reduce the overhead of taint tracking, it cannot eliminate it completely. As long as taint propagation is being done in-line with the application code, there will always be some degree of overhead involved with running those calculations. In order to eliminate that final cost, it's necessary to decouple the app's code from the instrumentation code and run the two in parallel. In a parallelized taint tracking model, the application thread does not maintain its own taint state. Instead, when an instruction is executed, the application thread records information about the instruction and sends it to a taint collector thread running on a different core. Using the information sent by the application thread, the collector thread then infers the changes made to the program, and updates its taint state appropriately. With this design, the slowdown of the main app thread is limited to the time necessary for communication, rather than the full overhead of taint propagation. These techniques have proven effective in native-code taint tracking workloads, as in ShadowReplica [8].

Unfortunately, while they work well on heavier workloads, software-only parallelization techniques like ShadowReplica become less effective as the analysis costs decrease, due to the fixed costs of communication. In the case of our analysis, where our propagation logic is small in comparison to a native taint tracking workload, that fixed communication cost may negate any benefits gained from moving the propagation off-core. This would seem to indicate that the maximum speed of our analysis has been reached - it can't be sped up any further in-line, and it can't be profitably moved out-of-line.

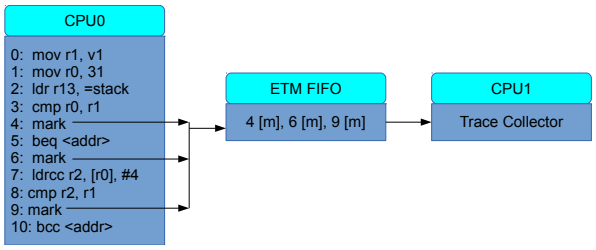

Figure 2: Instruction markers are added to the app code during ahead-of-time compilation. The markers generate ETM events, causing the PC and optional instruction data (m) to be saved in the FIFO. A taint collector core reads the FIFO and performs taint tracking.

However, there is still one technique available - if there were a way to cheaply communicate instruction-trace data off-core, then parallelization would become a viable option, even on workloads this small. Previous work has considered adding hardware support for this communication, and has shown it to be an effective technique [6]. Unfortunately, as no hardware implementing this support has ever become commercially available, using it becomes impossible in practice.

As it turns out, though, while no processors have ever been designed to have this support, there *have* been processors with hardware designed to solve the similar problem of collecting trace data for debugging – namely, the ubiquitous ARM processor. Recent ARM chips have shipped with a small unit called an Embedded Trace Macrocell included in each processor core. This unit is designed to collect information about an execution trace, compress it, and write it into an on-chip FIFO for later collection by an external JTAG programmer. [1] In the absence of a JTAG, however, the FIFO can also be read out from any core as an ordinary memory-mapped peripheral. The ETM has configurable filters to limit the trace data to desired instructions, and can collect the program counter, data address, and data value of each instruction.

Those capabilities are sufficient, given correct configuration and software support, to allow the ETM to be used as an effectively-free taint trace communications channel off-core. Consider Figure 2. The app's native code is modified during compilation to add instruction markers (`mark` in Figure 2) whose purpose is to generate ETM events. Upon an ETM event, the processor saves the current PC and instruction data (labeled as "m") into a FIFO. The taint collector core reads out the FIFO to infer the taint state of the monitored app.

This technique has significant benefits over pure-

software solutions. First and foremost, this technique can be implemented on nearly any ARM processor made in the last several years. As ARM processors control nearly all of the smartphone market, this means that hardware support will be widely available, even at its first launch. Second, this technique is incredibly lightweight - because the ETM is integrated directly into the processor core, messages can be sent in as little as one cycle. Unlike software techniques, synchronization is handled automatically - if the ETM's output FIFO fills up, the processor will stall, thus allowing the core reading the FIFO time to catch up. This means that there is no need for any communication or synchronization between the application core and the taint collector core, except under limited circumstances - for example, to flush the buffer and ensure there is no leak before committing a write. Lastly, because the event generation is controlled by a hardware peripheral, event sources can be toggled on and off at runtime without needing to modify the code that generates them. A disabled event would still execute the marker instruction, but the corresponding ETM event would not be generated. This could reduce the load on the taint-collection thread and yield even more energy savings.

One challenge to this approach is the energy cost of running two cores instead of one, which may negate any energy savings gained by reducing CPU overhead. In many respects, this is a similar problem to the pre-existing issue of sensor processing, which requires a small, but constant, amount of CPU power. Similar techniques can be used to mitigate the energy cost of taint processing - for instance, if the user is willing to accept deferred notification that a leak occurred, then taint-propagation events could be batched, and handled in short bursts at intervals. Another solution would be to do the processing on a smaller, low-power core, such as the low-power microcontrollers included with some TI chips [14], or ARM's more recent big.LITTLE designs that use low-power cores for lighter workloads [2].

4 Related Work and Conclusions

Taint tracking, a form of information flow control [19], is recognized as an important security primitive to control usage of sensitive data. However, researchers have quickly realized that it is an expensive primitive to implement, as evidenced by the plethora of projects aimed at optimizing taint tracking for speed and energy consumption [15, 18, 21, 25, 6, 8, 13, 24].

Software-only approaches to optimization include LIFT's [15] runtime binary techniques, where code paths not affected by tainted data are not taint tracked. This mechanism does not require any specialized hardware support, but is an inline technique, i.e. execution and analysis are interleaved resulting in slowdowns. Speck [13] uses speculative execution and spare cores to run analysis on a separate core thus decoupling execution and analysis; however, this results in application code running more than once, which is an unacceptable energy cost on a mobile device. We propose an architecture that uses commodity real-time tracing hardware combined with static analysis to accelerate taint tracking on mobile devices. Our mechanism is not inline, and does not require re-execution of application code.

Log based architectures [6, 18] introduce a microarchitectural extension to stream execution logs from one core to an analysis core. However, we note that, typically, these logs are very high-bandwidth, resulting in application stalls and reductions in speed. ShadowReplica [8] is a promising technique that uses offline analysis to reduce the amount of execution trace transferred between the execution core and analysis core, but does not use hardware acceleration. This technique uses static analysis and profile information to minimize the required trace bandwidth, by ensuring that the most likely cases result in little or no data being generated.

Hybrid analysis techniques have recently started to emerge [4, 17, 25] that exploit offline analysis and semantics of program units such as functions. We believe that such techniques can be sped up even further through the use of semantic information and hardware acceleration. TaintEraser [25] reasons about the taint behavior of functions statically. However, unlike our proposal, it does not use hardware acceleration.

In summary, while taint tracking is an important security primitive, it is not yet widely adopted on mobile devices due to performance and energy constraints imposed by such an environment. In this paper, we describe the architecture of a hybrid taint tracking system that leverages the unique features of mobile operating systems. A central insight is that we can use existing real-time instruction tracing support available on ARM processors to implement a decoupled taint tracking system, assisted by static analysis on Java code to reduce the overhead of analysis. An implementation is in progress.

Acknowledgements

The authors thank Fengguo Wei for his input on static analysis. We also thank the anonymous reviewers for their comments and suggestions.

This material is based upon work supported by the National Science Foundation under Grant Numbers 093629 and 1318722. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also thank General Motors for supporting part of this research.

References

[1] ARM. Embedded trace macrocell architecture specification (etmv1.0 to etmv3.5). `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html`, 2011. Online; accessed 9 Jan 2015.

[2] ARM. Ten things to know about big.little. `https://community.arm.com/groups/processors/blog/2013/06/18/ten-things-to-know-about-biglittle`, 2013. Online; accessed 20 Apr 2015.

[3] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 29.

[4] CHANG, W., STREIFF, B., AND LIN, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 39–50.

[5] CHEN, R. Non-classical processor behavior: How doing something can be faster than not doing it. `http://blogs.msdn.com/b/oldnewthing/archive/2014/06/13/10533875.aspx`, 2014. Online; accessed 9 Jan 2015.

[6] CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., AND VLACHOS, E. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2008), ISCA '08, IEEE Computer Society, pp. 377–388.

[7] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM 57*, 3 (2014), 99–106.

[8] JEE, K., KEMERLIS, V. P., KEROMYTIS, A. D., AND PORTOKALIDIS, G. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 235–246.

[9] LE GUERNIC, G. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE* (2007), IEEE, pp. 218–232.

[10] LE GUERNIC, G., BANERJEE, A., JENSEN, T., AND SCHMIDT, D. A. Automata-based confidentiality monitoring. In *Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues*. Springer, 2007, pp. 75–89.

[11] MOORE, S., AND CHONG, S. Static analysis for efficient hybrid information-flow control. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th* (2011), IEEE, pp. 146–160.

[12] NAIR, S. K., SIMPSON, P. N., CRISPO, B., AND TANENBAUM, A. S. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science 197*, 1 (2008), 3–16.

[13] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 308–318.

[14] OMAPPEDIA. Ducati for dummies. `http://omappedia.org/wiki/Ducati_For_Dummies`, 2011. Online; accessed 20 Apr 2015.

[15] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 135–148.

[16] RASTHOFER, S., ARZT, S., AND BODDEN, E. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)* (2014).

[17] ROCHA, B. P. S., CONTI, M., ETALLE, S., AND CRISPO, B. Hybrid static-runtime information flow and declassification enforcement. *IEEE Transactions on Information Forensics and Security 8*, 8 (2013), 1294–1305.

[18] RUWASE, O., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., CHEN, S., KOZUCH, M., AND RYAN, M. Parallelizing dynamic information flow tracking. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2008), SPAA '08, ACM, pp. 35–45.

[19] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[20] TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. Taj: effective taint analysis of web applications. *ACM Sigplan Notices 44*, 6 (2009), 87–97.

[21] VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. Flexitaint: A programmable accelerator for dynamic taint propagation. In *In 14th International Symposium on HighPerformance Computer Architecture (HPCA-14* (2008).

[22] WARTELL, R., ZHOU, Y., HAMLEN, K. W., KANTARCIOGLU, M., AND THURAISINGHAM, B. Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.

[23] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1329–1341.

[24] WEI, Z., AND LIE, D. Lazytainter: Memory-efficient taint tracking in managed runtimes. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (New York, NY, USA, 2014), SPSM '14, ACM, pp. 27–38.

[25] ZHU, D. Y., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 142–154.