Title
Beneath the Attack Surface

Permalink
https://escholarship.org/uc/item/9qw030qx

Author
Mowery, Keaton

Publication Date
2015

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, SAN DIEGO

Beneath the Attack Surface

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Keaton Mowery

Committee in charge:

Professor Hovav Shacham, Chair
Professor Sorin Lerner
Professor George Papen
Professor Stefan Savage
Professor Geoffrey M. Voelker

2015
Copyright
Keaton Mowery, 2015
All rights reserved.
The Dissertation of Keaton Mowery is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego
2015
“Time forks perpetually toward innumerable futures. 
In one of them I am your enemy.”
— **Jorge Luis Borges**
(1941)

Marco Polo imagined answering (or Kublai Khan imagined his answer) that 
the more one was lost in unfamiliar quarters of distant cities, 
the more one understood the other cities he had crossed to arrive there 
— **Italo Calvino**
(1972)
# TABLE OF CONTENTS

Signature Page .................................................................................................................. iii
Epigraph ............................................................................................................................... iv
Table of Contents .................................................................................................................. v
List of Figures ....................................................................................................................... viii
List of Tables ......................................................................................................................... xi
Acknowledgements ............................................................................................................ xii
Vita ........................................................................................................................................ xiv
Abstract of the Dissertation ............................................................................................... xvi

Introduction .......................................................................................................................... 1

Chapter 1  Fingerprinting Information in JavaScript Implementations ............................... 3
  1.1 Introduction ..................................................................................................................... 4
  1.2 JavaScript Performance Fingerprinting .................................................................... 8
    1.2.1 Methodology .......................................................................................................... 8
    1.2.2 Data Collection ...................................................................................................... 10
    1.2.3 Results .................................................................................................................. 13
    1.2.4 JavaScript Test Selection ...................................................................................... 21
  1.3 NoScript Whitelist Fingerprinting ........................................................................... 22
    1.3.1 Attack Methodology .............................................................................................. 23
    1.3.2 Prevalence of Testable JavaScript ....................................................................... 26
    1.3.3 Fingerprinting Speed ............................................................................................. 28
  1.4 Conclusions ................................................................................................................... 32

Chapter 2  Pixel Perfect: Fingerprinting Canvas in HTML5 ........................................... 34
  2.1 Introduction ..................................................................................................................... 34
  2.2 HTML5 and CSS3 ......................................................................................................... 39
    2.2.1 HTML5 Canvas ...................................................................................................... 39
    2.2.2 WebFonts .............................................................................................................. 41
    2.2.3 WebGL ................................................................................................................ 41
    2.2.4 Security Implications ......................................................................................... 42
  2.3 Experiments ................................................................................................................... 43
    2.3.1 Tests ..................................................................................................................... 43
    2.3.2 Infrastructure ....................................................................................................... 46
    2.3.3 Data Collection ................................................................................................. 47
2.4 Results .......................................................... 50
  2.4.1 Arial Font Rendering ..................................... 50
  2.4.2 WebFont Rendering ......................................... 55
  2.4.3 WebGL ...................................................... 58
  2.4.4 Comprehensive Fingerprinting ............................ 62
2.5 Defenses ....................................................... 63
2.6 Conclusions .................................................. 64
2.7 Data Characterization ....................................... 66

Chapter 3 Are AES x86 Cache Timing Attacks Still Feasible? .............. 70
  3.1 Introduction .................................................. 70
  3.2 Complete Mitigation ........................................ 72
    3.2.1 AES-NI .................................................. 73
    3.2.2 Multicore Processors ................................... 74
  3.3 Attack Outline .............................................. 75
  3.4 Modern Software Engineering ............................... 78
    3.4.1 Chromium Architecture ................................ 78
    3.4.2 Measurements .......................................... 79
  3.5 Prefetching .................................................. 80
  3.6 Cache Indexing ............................................. 83
    3.6.1 Attack Complexity ...................................... 85
  3.7 Conclusions ................................................ 86

Chapter 4 Welcome to the Entropics: Boot-Time Entropy in Embedded Devices 88
  4.1 Introduction ................................................. 88
    4.1.1 Related Work ........................................... 91
  4.2 Early Kernel Entropy ...................................... 93
    4.2.1 Genesis ................................................ 94
    4.2.2 Methodology .......................................... 95
    4.2.3 Results and Analysis .................................. 97
  4.3 Architectural Causes of Timing Variation .................... 115
    4.3.1 Clock domain crossing ................................ 116
    4.3.2 DRAM Access Latency ................................ 118
  4.4 DRAM Decay ................................................ 120
    4.4.1 Disabling Refresh ...................................... 122
    4.4.2 Decay .................................................. 122
    4.4.3 Experimental Setup .................................... 122
    4.4.4 Results ................................................. 123
    4.4.5 Extracting per-boot randomness from DRAM ............ 126
  4.5 PLL Lock Latency .......................................... 127
  4.6 Conclusions ................................................ 128

Chapter 5 Security Analysis of a Full-Body Scanner ...................... 132

vi
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Comparison of relative browser performance: Normalized test time vectors for different browsers on a single machine</td>
<td>14</td>
</tr>
<tr>
<td>1.2</td>
<td>Whitelist test page for google.com</td>
<td>25</td>
</tr>
<tr>
<td>1.3</td>
<td>Completion Time CDF for domain whitelist check (NoScript Off)</td>
<td>29</td>
</tr>
<tr>
<td>1.4</td>
<td>Completion Time CDF for domain whitelist check (NoScript On)</td>
<td>30</td>
</tr>
<tr>
<td>2.1</td>
<td>JavaScript code to render text on a canvas</td>
<td>40</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of text_arial and text_arial_px</td>
<td>43</td>
</tr>
<tr>
<td>2.3</td>
<td>An example of text_webfont and text_webfont_px</td>
<td>44</td>
</tr>
<tr>
<td>2.4</td>
<td>An example of text_nonsense</td>
<td>45</td>
</tr>
<tr>
<td>2.5</td>
<td>An example of webgl</td>
<td>46</td>
</tr>
<tr>
<td>2.6</td>
<td>13 ways to render 20px Arial</td>
<td>52</td>
</tr>
<tr>
<td>2.7</td>
<td>Difference maps for a group on text_arial</td>
<td>54</td>
</tr>
<tr>
<td>2.8</td>
<td>10 ways to render 12pt Sirin Stencil</td>
<td>57</td>
</tr>
<tr>
<td>2.9</td>
<td>Sirin Stencil with no Antialiasing</td>
<td>58</td>
</tr>
<tr>
<td>2.10</td>
<td>Original WebGL render and selected difference maps for Group 24</td>
<td>59</td>
</tr>
<tr>
<td>3.1</td>
<td>Access times per cache line after cache flush on Atom N270</td>
<td>82</td>
</tr>
<tr>
<td>3.2</td>
<td>Access times per cache line after cache flush on Atom N270</td>
<td>82</td>
</tr>
<tr>
<td>3.3</td>
<td>Address bits on Atom N270</td>
<td>82</td>
</tr>
<tr>
<td>3.4</td>
<td>Average access time for virtual addresses which should collide in the cache.</td>
<td>84</td>
</tr>
<tr>
<td>4.1</td>
<td>Histogram of cycle counts for Test 4 on 4 Raspberry Pis</td>
<td>100</td>
</tr>
<tr>
<td>4.2</td>
<td>Histogram of cycle counts for Test 5 on 4 Raspberry Pis</td>
<td>101</td>
</tr>
<tr>
<td>4.3</td>
<td>Histogram of cycle counts for Test 36 on 4 Raspberry Pis</td>
<td>101</td>
</tr>
</tbody>
</table>
Figure 5.8. Attacking Privacy .................................................. 156
Figure 5.9. Naïve Evaluation ................................................. 161
LIST OF TABLES

Table 1.1. Fingerprint collected for classification: OS and browser breakdown 12
Table 1.2. Browser Detection Results ............................................. 16
Table 1.3. Pairwise distances of Chrome major version fingerprints .......... 17
Table 1.4. Detected Operating Systems under Firefox 3.6 ..................... 18
Table 1.5. Detected CPU Architecture under Firefox 3.6 ...................... 20
Table 2.1. Selected groupings of identical WebGL renders. Each group corre-
sponds to a single pixmap. .......................................................... 60
Table 2.2. Groups of identical WebGL renders. Number in parentheses indicate
device IDs, where appropriate....................................................... 67
Table 4.1. Per-Test Distribution Entropy, in bits ............................... 103
ACKNOWLEDGEMENTS

First, I’d like to thank my family — for always supporting me in this journey, and for never wondering too loudly if grad school was the right path.

Next, I must thank my most excellent advisor, Hovav Shacham. I’ve worked with Hovav since I began this research journey, and he’s taught me everything I’ve needed to finish this Ph.D.

Corey Singleton and Chris Comfort, in particular, were utterly indispensable for my experiments into X-ray physics and their health effects. They patiently answered all my silly questions and were always happy to loan out their toys (occasionally indefinitely!). Tom Wypych, similarly essential, was always able to find exactly the thing we needed, from time on a CnC mill to the phone number of a company selling plastic explosive simulants. I’d also like to thank Daniel Scanderbeg at UCSD Radiation Oncology for the 2-year loan of a radiological torso phantom.

I’d also like to thank each one of my coauthors: Marc Andrysco, Dillon Bogenreif, Professor Stephen Checkoway, Chris Comfort, Arel Cordero, Professor J. Alex Halderman, Professor Ranjit Jhala, Theron Ji, Professor Chris Kanich, Sriram Keelvedhi, David Kohlbrenner, Professor Sorin Lerner, Professor Sarah Meiklejohn, Eric Rescorla, Professor Stefan Savage, Corey Singleton, Professor Steve Swanson, Alan Tsai, Professor David Wagner, Michael Wei, Eric Wustrow, Tom Wypych, and Professor Scott Yilek. I couldn’t have done it without you all.

I’d also like to thank a few close friends I’ve made along the way — without you, I wouldn’t have made it this far: Joe Auricchio, Neha Chachra, Rowan Dorland, David Kohlbrenner, and Michelle Olofson.

Lastly, everyone I’ve shared CSE 3140 with deserves recognition, not least for making it the most requested lab around. So, thanks: Neha Chachra, Mike Conley, Joe DeBlasio, Tristan Halvorson, Danny Huang, Chris Kanich, David Kohlbrenner, Wilson

Chapter 1, in part, is a reprint of the material as it appears in Web 2.0 Security and Privacy (W2SP) 2011. Mowery, Keaton; Bogenreif, Dillon; Yilek, Scott; Shacham, Hovav, 2009. The dissertation author was a primary investigator and the primary author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in Web 2.0 Security and Privacy (W2SP) 2012. Mowery, Keaton; Shacham, Hovav, 2012. The dissertation author was a primary investigator and the primary author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in ACM Cloud Computing Security Workshop (CCSW) 2012. Mowery, Keaton; Keelveedhi, Sriram; Shacham, Hovav, 2012. The dissertation author was a primary investigator and the primary author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in IEEE Security and Privacy (Oakland) 2013. Mowery, Keaton; Wei, Michael; Kohlbrenner, David; Shacham, Hovav; Swanson, Steven, 2013. The dissertation author was a primary investigator and the primary author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in USENIX Security 2014. Mowery, Keaton; Wustrow, Eric; Wypych, Tom; Singleton, Corey; Comfort, Chris; Rescorla, Eric; Checkoway, Stephen; Halderman, J. Alex; Shacham, Hovav, 2014. The dissertation author was a primary investigator and the primary author of this paper.
VITA

2009 Bachelor of Science in Electrical Engineering and Computer Science  
University of California, Berkeley

2009–2012 Research Assistant  
University of California, San Diego

2012 Master of Science in Computer Science  
University of California, San Diego

2012–2015 Research Assistant  
University of California, San Diego

2015 Doctor of Philosophy in Computer Science  
University of California, San Diego

PUBLICATIONS


Computer systems are often analyzed as purely virtual artifacts, a collection of software operating on a Platonic ideal of a computer. When software is executed, it runs on actual hardware: an increasingly complex web of analog physical components and processes, cleverly strung together to present an illusion of pure computation. When an abstract software system is combined with individual hardware instances to form functioning systems, the overall behavior varies subtly with the hardware. These minor variations can change the security and privacy guarantees of the entire system, in both beneficial and harmful ways. We examine several such security effects in this dissertation.
First, we look at the fingerprinting capability of JavaScript and HTML5: when invoking existing features of modern browsers, such as JavaScript execution and 3-D graphics, how are the results affected by underlying hardware, and how distinctive is the resulting fingerprint?

Second, we discuss AES side channel timing attacks, a technique to extract information from AES encryption running on hardware. We present several reasons why we were unable to reproduce this attack against modern hardware and a modern browser.

Third, we examine positive uses of hardware variance: namely, seeding Linux’s pseudorandom number generator at kernel initialization time with true entropy gathered during early boot. We examine the utility of these techniques on a variety of embedded devices, and give estimates for the amount of entropy each can generate.

Lastly, we evaluate a cyberphysical system: one which combines physical processes and analog sensors with software control and interpretation. Specifically, we examine the Rapiscan Secure 1000 backscatter X-ray full-body scanner, a device for looking under a scan subject’s clothing, discovering any contraband secreted about their person. We present a full security analysis of this system, including its hardware, software, and underlying physics, and show how an adaptive, motivated adversary can completely subvert the scan to smuggle contraband, such as knives, firearms, and plastic explosives, past a Secure 1000 checkpoint. These attacks are entirely based upon understanding the physical processes and sensors which underlie this cyberphysical system, and involve adjusting the contraband’s location and shape until it simply disappears.
Introduction

In computer security, the “attack surface” of a system is the portion of that system which is reachable by an attacker, and thus is potentially vulnerable. Usually these attacks involve modifying the system in some way, or perhaps convincing it to divulge secret data, but the attacker will almost always be working outside the assumptions of the original system designer — either by sending carefully crafted inputs, claiming to have authorization, or some other technical means of attack.

The system operator has several options to reduce their attack surface. First, they could harden the attack surface, through careful inspection and modification of each component in the attack surface. Or, they can reduce the attack surface, either by moving vulnerable components behind other, more secure pieces, or removing them altogether.

In the following chapters, despite occasionally taking the role of attacker, we will dismiss the attack surface completely. All of our inputs are legal; the produced results well within those expected by the system designers. Nonetheless, through preparation and analysis, we are able to achieve several different security results — learning things thought hidden, hiding things thought visible, and producing useful data from nothing.

To achieve this, we can examine these systems as they actually are run: an artifact composed of software processes running on actual, physical hardware. Hardware should produce the same logical result every time a computation is run, but each individual piece of hardware will achieve this result in its own way. With effort, these differences can be teased out, and can leak information about the combination of software and hardware.
When I proposed this thesis, one committee member called it “eclectic”. And it is! We will discuss the internal workings of web browsers and their JavaScript engines and what it means for privacy online, how to amplify the slight effects of physical microprocessor layout to produce random numbers, what modern software engineering practices mean for a class of hardware-based attacks on encryption, and the implications of the physical interactions of X-rays and matter on airport security.

Outline

In Chapters 1 and 2, we examine the browser: if you let a website run basic JavaScript code, render HTML5 pages, and give it access to a basic timer, what can it learn about you?

Next, in Chapter 3, we discuss cache timing attacks against the AES block cipher. Cache timing attacks are a classic example of an attack against software running on an actual processor, but in this case, we were unable to successfully mount a cache timing attack against the TLS AES implementation in Google Chrome. This chapter covers our attempts and the reasons we believe AES cache timing attacks have become impossible against real-world software.

In Chapter 4, we present a positive security result: a new technique for generating entropy upon boot on resource-starved embedded devices. This entropy can be added to the operating system’s Psuedo-Random Number Generator (PRNG) before the kernel boots, closing a systemic flaw uncovered by Nadia Heninger et al. [59] in 2012.

Lastly, in Chapter 5, we examine a deployed cyberphysical security system: the Rapiscan Secure 1000 x-ray backscatter machine, and show how an adaptive, motivated adversary can completely subvert the scan to smuggle contraband, such as knives, firearms, and plastic explosives, past a Secure 1000 checkpoint.
Chapter 1

Fingerprinting Information in JavaScript Implementations

To date, many attempts have been made to fingerprint users on the web. These fingerprints allow browsing sessions to be linked together and possibly even tied to a user’s identity. They can be used constructively by sites to supplement traditional means of user authentication such as passwords; and they can be used destructively to counter attempts to stay anonymous online.

In this chapter, we identify two new avenues for browser fingerprinting. The new fingerprints arise from the browser’s JavaScript execution characteristics, making them difficult to simulate or mitigate in practice. The first uses the innate performance signature of each browser’s JavaScript engine, allowing the detection of browser version, operating system and microarchitecture, even when traditional forms of system identification (such as the user-agent header) are modified or hidden. The second subverts the whitelist mechanism of the popular NoScript Firefox extension, which selectively enables web pages’ scripting privileges to increase privacy by allowing a site to determine if particular domains exist in a user’s NoScript whitelist.

We have experimentally verified the effectiveness of our system fingerprinting technique using a 1,015-person study on Amazon’s Mechanical Turk platform.
1.1 Introduction

A unique fingerprint that identifies a browser and its user has many uses on the Web. Used constructively, such a fingerprint allows sites to recognize returning users both between visits and during a visit, for example to avoid asking for login credentials for every action that requires authentication, or to detect account hijacking and impersonation. Used maliciously, such a fingerprint may allow an attacker to track a user between sites (even when those sites do not cooperate with the attacker) and to identify users who use privacy-enhancing technologies such as Tor.

The traditional means for a user to identify himself to a service is the password. With services and user browsing data moving to cloud-hosted infrastructure, it is increasingly the case that large-scale data compromise makes it hard to have a firm hold on customer identity. Since users reuse passwords at multiple sites, this is true even when a site impeccably secures its own login database: The recent Gawker compromise allowed hackers to compromise many Twitter accounts that reused the same login credentials.

Accordingly, for many sites, and especially those (such as online banking) where fraud is a concern, identifying users by password alone is not sufficient. Increasingly, such sites employ additional means to identify the user and the system she is using. Hardware tokens are appropriate in certain settings, but their cost and inconvenience form a barrier to ubiquitous use. For such sites, browser and user fingerprints form an attractive alternative to hardware tokens. On the (reasonable) assumption that typical users will usually log in from a single machine (or, at most, a handful), JavaScript fingerprinting allows sites to harden passwords against account hijacking with a minimum of user inconvenience.

Destructively, fingerprints can also be used to identify and track users, even when those users wish to avoid being tracked. The most familiar browser fingerprinting technology is the cookie, a client-side datastore partitioned by domain and path. Cookies
installed by third-party content included on sites users visit allow users to be tracked between sites. Modern browsers provide many other client-side datastores that can provide cookie-like functionality [81]. The browser history and file cache are effectively unpartitioned by domain and can be queried by timing or DOM APIs [69], and can be used as a fingerprint [78]. Information disclosed by the browser in headers, through JavaScript APIs, and through plugins can provide a fingerprint with more than 18 bits of effective entropy [94, 43]. Even quirks of the browser’s JavaScript handling and supported DOM features can be used as a fingerprint [47]. These techniques can be used in concert by sites, yielding a fingerprint that can uniquely identify users and facilitate their being tracked.

Our results

We describe two new techniques for fingerprinting browsers, both making use of the JavaScript interpreter. The fingerprinting information we obtain can be used alongside previously known fingerprinting technologies, such as those explored in EFF’s Panopticlick project, to enable user device identification and supplement user passwords (in a constructive application) or to reduce the anonymity set of Web users even further (in a destructive application).

Our first technique times the performance of different operations in the core of the JavaScript language. Unlike previous fingerprinting techniques based on JavaScript [47, 43], ours depends not on functionality differences, but performance differences between browsers. This means that it can be used even when limited functionality is made available to JavaScript. Also, because our technique takes advantage of the execution timing differences between instruction sequences produced by the browser’s JavaScript JIT, it can distinguish not only browser versions but also microarchitectural features not normally exposed to JavaScript. As browsers compete largely on JavaScript performance, any countermeasure that involves slowing JavaScript execution or preventing
accurate timing will be unacceptable to vendors, making this fingerprinting technique very robust.

Our second technique allows the attacker to query entries in the user’s NoScript whitelist. NoScript [93] is a Firefox extension that allows users to whitelist or blacklist domains for JavaScript execution. When a user visits a page, scripts will execute only if the site’s domain is whitelisted. Moreover, if page includes a script from a second domain, then that script will execute only when that script-provider domain is also whitelisted. Many sites do not function properly when JavaScript is disabled, so users customize their NoScript whitelist for their favorite sites. We show that a site can include a script from another domain to determine whether that other domain is whitelisted; scripts appropriate for including can be automatically found for a large fraction of the Alexa Top 1000 sites. When tested in aggregate, the domains found in the NoScript whitelist constitute a fingerprint of a user’s preferred sites and habits, leaking valuable information about their preferences to attackers. While this attack targets a particular browser extension (which may not be installed by every fingerprinted user), its existence illustrates the possibility of arbitrary websites probing for the existence and internal state of browser extensions.

Applications and threat model

For constructive applications, we consider an attacker who has stolen a user’s login credentials (via malware or phishing) and is attempting to impersonate the user. Such an attacker can study the victim’s system configuration and could mimic the user-agent string, but may not be able to duplicate the victim’s hardware and software configuration. Under this model, the attacker may impersonate the user in any self-reported fingerprint scheme (such as the Panopticlick approach of accessing user agent, available plugins, and other data through JavaScript APIs). When creating a fingerprint directly from raw JavaScript performance, however, the attacker will be unable to replicate
the user’s timings for every conceivable script. This unforgeability property provides an inherent advantage over other browser fingerprinting schemes.

For destructive applications, we consider browsers’ “private browsing” modes [4] or the Tor network [40] together with the Torbutton extension for Firefox [110]. For the former, we adopt the Web attacker model of Aggarwal et al. [4]; for the latter, we also consider a rogue exit node model; unlike a Web attacker, a rogue Tor exit node can inject content that appears to come from sites the attacker doesn’t control — including sites on the default NoScript whitelist such as addons.mozilla.org.

Related work

When operations depend on secret values, the time it takes to complete them often leaks information useful to the attacker. Timing side channels have been used to attack several cryptosystems, notably a remote timing attack on RSA decryption in SSL [25]. In a Web context, timing has been used against clients to determine whether content was in the cache [45] and against servers to learn information such as the number of items in a user’s shopping cart [19]. Clock skew, which is correlated with processor activity, has been used to identify Tor hidden services [102]. Finally, we observe that other sources of information, including summaries of the Web traffic they generate, can be used to fingerprint browsers [152].

Our NoScript whitelist fingerprint is closely related to history sniffing [69, 73]. The different decoration applied by browsers to visited and unvisited links, together with ways for sites to determine the decoration applied to page elements, makes it possible for attackers to determine whether a user’s visiting their site had previously visited any URL of interest. This technique, known since at least 2002 [29], can be used to interrogate tens of thousands of URLs per second [74]. Whereas history sniffing applies to specific URLs, NoScript whitelist sniffing applies to domains; whereas history entries are set
automatically whenever a user visits a page, NoScript whitelists are changed manually by users. These two differences mean that NoScript whitelist sniffing provides less information than history sniffing; the latter difference may mean that the fingerprint it provides is more stable over time. Moreover, with browsers deploying fixes against traditional history-sniffing techniques (see, e.g., [26, 12] for Firefox), the amount of information available to attackers from history sniffing may be reduced, making other techniques more attractive by comparison.

1.2 JavaScript Performance Fingerprinting

As browsers advance and compete on features and usability, one of the largest areas of development is the JavaScript engine. Benchmark suites, such as V8 and SunSpider, are created to measure scripting speed. As developers apply clever approaches such as just-in-time script compilation, the JavaScript engines improve. These improvements are not uniform, however: different approaches yield different payoffs on certain types of script. The process of incremental improvement produces a tell-tale signature, detectable simply through timing the execution of otherwise innocuous JavaScript. By leveraging these discrepancies, timing information can be used to fingerprint the host machine.

1.2.1 Methodology

Utilizing the SunSpider and V8 JavaScript benchmarks along with custom code, we constructed a suite of 39 individual JavaScript tests. To create a fingerprint, we measure the time in milliseconds to complete each test in our benchmark suite. This produces a 39-dimensional vector, which characterizes the performance of the tested machine.

Naturally, differences in processor architecture and clock speed impact these timings significantly. Many processors also support CPU throttling, which dynamically slows the processor during periods of low utilization. Similarly, benchmark completion
times may be affected if external (non-browser) processes cause significant load. To remove these effects from our measurements, we normalize the timing vector before attempting classification.

Oddly, single-test times, even during an otherwise-idle browser session, can vary widely. This is possibly due to other scripts running in the browser, JavaScript garbage collection, or even the timing infrastructure used by our benchmarks. We take several steps to minimize the effects of this variance. First, we add an 800 ms delay between the end of one test and the start of another, allowing time for any cleanup and execution of scripts running in other windows of the session. Secondly, we run each test five times, and take the minimum positive time for each test. Intuitively, while the browser may be slowed, it will never be induced to perform better than its maximum speed. Also, our JavaScript timing framework reported that an individual test time took 0 ms an appreciable fraction of the time, even dipping to −1 ms once or twice (which is clearly impossible). Running each test multiple times smooths out these glitches and random browser slowdowns, in effect decreasing the variance of the final test vector. While these techniques increase the reliability of our tests, they do impose a penalty in terms of execution time.

Our benchmark suite takes 190.8 s to complete on Firefox 3.6. However, due to the per-test 800 ms timeout, the test suite spends approximately 156 s sleeping. By using a smaller timeout, the total time could be reduced significantly. Also, we did not experiment with removing tests — our results may be achievable with less than 39 tests.

(More generally, while we have used off-the-shelf JavaScript benchmark scripts to demonstrate the feasibility of our approach, we believe that custom-written scripts targeting specific JavaScript engine revisions and microarchitectural features would execute more efficiently and provide an even more effective fingerprint than our prototype.)

Once we have a fingerprint vector for a given configuration, we need some way of classifying a test vector in order to infer facts about the user’s setup. To do so, we utilize
a simple method: match the fingerprint vector to a representative vector corresponding to a known configuration. From our data, we generate fingerprint vectors of several leading browsers, and use these to classify the browser used to execute the benchmark code.

**Optimization**

One of the largest weaknesses in our approach is that the fingerprinting time is very large — usually over 3 minutes. Much of this time is spent on ways to reduce timing jitter: an 800 ms pause is inserted between each test, and the entire benchmark suite is executed five separate times. As noted recently by the Google Chrome team [14], modern browsers complete most SunSpider benchmarks extremely quickly (under 10 ms), and so a random delay of just a few milliseconds can appear as a momentous difference in test times. By running each individual test many times in a row (Google chose 50 repetitions for their SunSpider variant), timing variance can be greatly reduced. As a bonus, letting each test run longer will mitigate the JavaScript timing idiosyncrasies that result in negative or zero elapsed times. Taken together, these facts suggest that timing many runs of each individual benchmark test (instead of timing each test many times) will reduce timing variance significantly. Heavy-handed measures such as long inter-test delays and gratuitous repetition can be discarded, thereby eliminating our largest sources of delay and shortening our fingerprinting time considerably.

**1.2.2 Data Collection**

We collected JavaScript fingerprints for 1015 user configurations, where each configuration is an operating system, browser, hardware, and benchmark performance tuple. To achieve data collection on this scale, we conducted a survey on the Amazon Mechanical Turk marketplace, paying users a small monetary sum to report their CPU model, clock speed, RAM, and operating system and to allow our JavaScript to run. We
also record the browser-reported User Agent string, which we treat as ground truth (i.e., we assume that user agents are not forged).

The operating systems reported by our users include variants of Windows (7, Vista, XP, NT 4.0), OS X (10.6, 10.5, 10.4) and Linux (Ubuntu, Mint, generic). As for browsers, we observed usage of Chrome, Firefox, Internet Explorer, Opera, Safari, and SeaMonkey. The exact breakdowns are given in Table 1.1.

Intel processors underly a vast majority of our participants’ systems, with the most popular being Core 2 (395), Pentium Dual Core (137), and Pentium 4 (123). Other common platforms include Core i3, Core, Atom, Athlon 64, Core i5, and Athlon II.

Overall, our results appear to contain a representative sample of the browsers, operating systems and hardware configurations in use on the Web today.

Data Quality

Our Mechanical Turk survey presented users with three free-form text entry boxes, along with drop-down menus to report operating system and number of CPU cores. We also provide step-by-step instructions, with example screenshots, detailing how to obtain the relevant information for both Windows and OS X.

Nevertheless, improperly filled-out surveys comprised a non-trivial portion of responses. Each of our 1015 samples was hand-verified, and we include every response for which we could decipher a distinct CPU, clock speed, core count, and RAM. We received an extra 185 submissions which did not measure up to this standard and were excluded from further consideration. A fairly reasonable (but unhelpful) answer was “Celeron”, as this does not specify a particular CPU model. We accepted their response (and paid the user), then excluded the result later in the process. Most of the rejections, however, were unusable: “Intel” and “x86” were very common. Some of the authors’ favorite submissions for CPU architecture included “Normal”, “HP Compaq LE1711”
Table 1.1. Fingerprints collected for classification: OS and browser breakdown

<table>
<thead>
<tr>
<th></th>
<th>Windows 7</th>
<th>Windows Vista</th>
<th>Windows XP</th>
<th>Windows NT 4.0</th>
<th>OS X 1.70.6</th>
<th>OS X 1.70.7</th>
<th>OS X 1.70.4</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome 2.0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chrome 3.0</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chrome 4.0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chrome 5.0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chrome 6.0</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chrome 7.0</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chrome 8.0</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>Chrome 9.0</td>
<td>98</td>
<td>25</td>
<td>87</td>
<td>-</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>Chrome 10.0</td>
<td>74</td>
<td>25</td>
<td>68</td>
<td>-</td>
<td>8</td>
<td>1</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Chrome 11.0</td>
<td>6</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Firefox 2.0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Firefox 3.0</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Firefox 3.1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Firefox 3.5</td>
<td>6</td>
<td>2</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Firefox 3.6</td>
<td>162</td>
<td>45</td>
<td>190</td>
<td>-</td>
<td>13</td>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Firefox 4.0</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IE 7.0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IE 8.0</td>
<td>34</td>
<td>6</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IE 9.0</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Opera 9.64</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Safari 4.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Safari 4.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Safari 5.0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>21</td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SeaMonkey 2.0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
(an HP LCD monitor), and two separate participants who reported their architecture as “von Neumann”.

Furthermore, among our 1015 accepted data points, 47 of them self-report a different operating system than their HTTP user agent claims. We did not investigate further, but note that this does suggest that some valid-looking but incorrect data might have been submitted by misunderstanding users. However, we ignore these effects and treat every report as correct.

1.2.3 Results

Browser Detection

First, we demonstrate the feasibility of determining browser version through JavaScript performance. To do so, we ran our test suite on a single computer in a variety of browsers. The test machine consisted of an Intel Core 2 Duo processor at 2.66 GHz with 4 GB of RAM. We created test vectors for Firefox 3.6.3, Internet Explorer 8.0, Opera 10.51, and Safari 4.0.5. Internet Explorer was tested on Windows 7, while the remaining three browsers were run on Mac OS X 10.6.3. Each browser returns a 39-dimensional normalized vector of test times, a subset of which are presented in Figure 1.1. Clearly, significant differences exist in the JavaScript performance profile of each browser, which we can exploit to recognize a browser’s unique JavaScript execution signature.

To determine the effectiveness of general browser detection via JavaScript performance, we use the survey-supplied labels to generate browser fingerprint vectors. For each browser under test, we average the first ten survey vectors we received, using 38 tests as vector elements (test generation and selection is discussed in Section 1.2.4). The resulting vector is the first-level browser fingerprint.

While the top-level fingerprints are generally reliable, in our experience it creates unnecessary misclassification errors between minor versions of Firefox (namely, 3.6 and
Figure 1.1. Comparison of relative browser performance: Normalized test time vectors for different browsers on a single machine

To combat this, we also create a second-level fingerprint, consisting of 37 tests, specifically for finer-grained Firefox classification. Since this fingerprint is only applied if the first-level classification indicates a browser in the Firefox family, we can strip out unneeded tests and produce fingerprint vectors to distinguish between minor variations in the SpiderMonkey JavaScript engine. This pattern of successively more precise tests can produce far better results than a single-level classification alone.

Our data set provides enough benchmarks to create fingerprint vectors for various major versions of Chrome (2.0 through 11.0) and Internet Explorer (7, 8, 9), along with minor versions for Firefox (2.0, 3.0, 3.1, 3.5, 3.6, 4.0b) and Safari (4.0, 4.1, 5.0). We also have both an Opera 9.64 and a SeaMonkey 2.0 sample. Our methods are sufficiently robust to reliably detect the distinct JavaScript execution profile for each of these 23 browser versions. Further work might be able to distinguish between even smaller browser revisions, if those revisions included changes to the JavaScript engine.

To classify an unknown configuration, we simply execute the test suite which creates the 39-element fingerprint vector. Classification then becomes a simple matter of comparing against browser fingerprints and choosing the nearest match, using the Eu-
clidean distance metric. The results of classification for our 1015 samples can be found in Table 1.2. Notably, we correctly classify 810 browsers, for an overall accuracy of 79.8%.

Our classification results include the reference vectors for each browser, i.e., the first ten samples from which we create the browser fingerprint vector. We use this approach mainly due to the relative excess of small categories in our data set: 15 browser versions contain less than 10 data points. For these browsers, we generate the test vector using every available sample. To demonstrate browser differentiation, then, we have no other samples with which to perform proper classification, so we classify the reference vectors themselves and see if they match their browser vector or not (note that an IE 7.0 sample is misclassified, even though the reference vector is made of itself and one other vector!). By keeping these categories, we also gain the benefit that vectors in the larger browser pools are compared against the smaller versions, allowing for misclassifications if the browsers’ performance characteristics are too similar. For consistency, we include the reference vectors of the larger categories in our classification results as well.

Examining our results in detail, 176 of 205 misclassifications occur between major versions of Chrome, with a Chrome detection correctness of 62.5%. We ascribe these failures to two major features. First, Chrome’s auto-updater continuously moves most users along the upgrade path. We acquired less than 5 fingerprint vectors for Chrome 2.0 through 7.0, which reduces our confidence in the overall major version fingerprint. Secondly, Chrome’s aggressive release schedule means that only about 6 months separated the release of Chrome 6.0 and Chrome 10.0. Our experiments indicate that JavaScript development continued over those six months, but not enough to reliably distinguish between versions using our test suite. This pattern of incremental improvement can be seen in Table 1.3, which displays the pairwise distance between Chrome fingerprint vectors.

Overall, our methodology is extremely robust for browser family detection (Chrome, Firefox, etc), with a correctness rate of 98.2%.
Table 1.2. Browser Detection Results. Columns correspond to classification, rows represent actual version (as reported by user agent)

|               | Chrome 2.0 | Chrome 3.0 | Chrome 4.0 | Chrome 5.0 | Chrome 6.0 | Chrome 7.0 | Chrome 8.0 | Chrome 9.0 | Chrome 10.0 | Chrome 11.0 | Firefox 2.0 | Firefox 3.0 | Firefox 3.1 | Firefox 3.5 | Firefox 3.6 | Firefox 4.0 | IE 7.0 | IE 8.0 | IE 9.0 | Opera 9.64 | Safari 4.0 | Safari 4.1 | Safari 5.0 | SeaMonkey 2.0 |
|---------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|----------|--------|--------|------------|-----------|-----------|------------|--------------|
|               | 1 | - | - | - | - | - | - | - | - | - | - | - | - | - | 1 | - | - | 1 | - | - | - | - | 1 |
|               | - | 3 | - | - | - | - | - | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - |
| Chrome 4.0    | - | - | 1 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 5.0    | - | - | 4 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 6.0    | - | - | - | 3 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 7.0    | - | - | - | 3 | - | 1 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 8.0    | - | - | - | 1 | 5 | 7 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 9.0    | - | - | - | 1 | 4 | 6 | 17 | 11 | 17 | 1 | 8 | 4 | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 10.0   | - | - | 1 | - | 1 | 6 | 1 | 25 | 79 | 69 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Chrome 11.0   | - | - | - | - | - | - | 1 | 4 | 8 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Firefox 2.0   | - | - | - | - | - | - | - | - | - | - | - | - | 5 | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Firefox 3.0   | - | - | - | - | - | - | - | - | - | - | - | - | - | 16 | - | - | - | - | - | - | - | - | - | - | - | - |
| Firefox 3.1   | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | - | - | - |
| Firefox 3.5   | - | - | - | - | - | - | - | - | 21 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Firefox 3.6   | - | - | - | - | - | - | - | - | 3 | - | 7 | 403 | 6 | - | - | - | - | - | - | - | - | 1 | 3 | - | - |
| Firefox 4.0   | - | - | - | - | - | - | - | - | - | - | - | - | 3 | - | - | - | - | - | - | - | - | - | - | - | - | - |
| IE 7.0        | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 1 | 1 | - | - | - | - | - | - | - | - | - |
| IE 8.0        | - | - | - | - | - | - | - | - | - | 3 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| IE 9.0        | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 3 | - | - | - | - | - | - | - | - | - | - | - |
| Opera 9.64    | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Safari 4.0    | 1 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Safari 4.1    | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 1 | - | - | - | - |
| Safari 5.0    | - | - | - | - | - | - | - | - | - | 4 | - | - | - | - | - | - | - | - | - | - | - | - | 2 | 28 | - | - | - |
| SeaMonkey 2.0  | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 1 | - | - | - |

|               | 8 | 9 | 10 | 11 | 2 | 3 | 3.1 | 3.5 | 3.6 | 4 | 7 | 8 | 9 | 9.64 | 4 | 4.1 | 5 | 2 |
Table 1.3. Pairwise distances of Chrome major version fingerprints

<table>
<thead>
<tr>
<th></th>
<th>6.0</th>
<th>7.0</th>
<th>8.0</th>
<th>9.0</th>
<th>10.0</th>
<th>11.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome 6.0</td>
<td>0.00</td>
<td>0.18</td>
<td>0.19</td>
<td>0.17</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Chrome 7.0</td>
<td>0.18</td>
<td>0.00</td>
<td>0.09</td>
<td>0.16</td>
<td>0.25</td>
<td>0.24</td>
</tr>
<tr>
<td>Chrome 8.0</td>
<td>0.19</td>
<td>0.09</td>
<td>0.00</td>
<td>0.17</td>
<td>0.27</td>
<td>0.26</td>
</tr>
<tr>
<td>Chrome 9.0</td>
<td>0.17</td>
<td>0.16</td>
<td>0.17</td>
<td>0.00</td>
<td>0.17</td>
<td>0.18</td>
</tr>
<tr>
<td>Chrome 10.0</td>
<td>0.25</td>
<td>0.25</td>
<td>0.27</td>
<td>0.17</td>
<td>0.00</td>
<td>0.09</td>
</tr>
<tr>
<td>Chrome 11.0</td>
<td>0.25</td>
<td>0.24</td>
<td>0.26</td>
<td>0.18</td>
<td>0.09</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Operating System Detection

In the previous subsection, we described techniques for fingerprinting browser versions via JavaScript performance. In this subsection, we extend our techniques to fingerprint operating system versions. This is more difficult to do, but also provides information that is more difficult to gather reliably from adversarial users by means of the User Agent string or other forms of browser self-reporting. Indeed, several alternative methods exist for detecting browser version, such as inspection of browser-specific JavaScript capabilities, whereas scripts must rely on browser self-reporting for such platform details as operating system version or underlying architecture.

The effects of operating system on JavaScript performance are quite small. In fact, they are vastly overshadowed by differences in the JavaScript engines themselves. To combat this, operating system detection is only feasible within a particular browser version. We chose to examine the effects of operating system on Firefox 3.6, as it was reliably detectable and had the most cross-platform data points in our survey (397 on Windows, 19 on OS X, and 8 on Linux). We followed the same procedure as in browser detection: using the same 38 tests from the first ten samples in each category to form the fingerprint, then using Euclidean distance for classification. The results of this classification are in Table 1.4.
Table 1.4. Detected Operating Systems under Firefox 3.6

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>OS X 10.4</th>
<th>OS X 10.5</th>
<th>OS X 10.6</th>
<th>Windows 7</th>
<th>Windows Vista</th>
<th>Windows XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>OS X 10.4</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OS X 10.5</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OS X 10.6</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Windows 7</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>76</td>
<td>68</td>
<td>17</td>
</tr>
<tr>
<td>Windows Vista</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>17</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>Windows XP</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>51</td>
<td>80</td>
<td>51</td>
</tr>
</tbody>
</table>

Notably, while we lack a significant number of Linux and OS X samples, our Windows identification rate is 98.5%, and we did not misclassify a single OS X example, demonstrating that operating system detection is quite possible through JavaScript benchmarking. Also, given the even distribution across successive Windows versions, these results indicate that detecting versions within an operating system family through JavaScript is difficult. Further targeted work is needed in this area.

Architecture Detection

Continuing our exploration of Firefox 3.6, we examine the effects of processor architecture on browser performance. In our data set, the major architectures running Firefox 3.6 are Core 2 (150), Pentium Dual Core (51), Pentium 4 (53), Core i3 (26), and Athlon 64 (22).

Our detection methodology for architecture differs from our previous classification strategies. We use a 1-nearest neighbor strategy, based on the same 38-test benchmark vector used in first-level browser classification. (We will discuss the reason for the change shortly.) While 1-nearest neighbor gives reasonable results, it performs very badly on underrepresented platforms with very few samples. Therefore, we exclude any architecture with 5 samples or fewer from consideration.
The results from the classification can be found in Table 1.5. Overall, we achieve a 45.3% classification success rate.

Examining the results in detail, we see clusters of misclassifications across similar processors. For example, processors sold under the “Pentium Dual Core” name are detuned versions of Core or Core 2 chips, and over half (71 of 120) of our misclassifications occur between the Core 2 and Pentium Dual Core categories. Core i3 and Core i5 processors can be extremely similar, with Intel Clarkdale and Arrandale processors being sold under both those names. Our Pentium D samples are mainly misclassified as Pentium 4, which we posit is due both to the minimal number of Pentium D examples as well as the design similarity between those processors. Our survey procedures collected manufacturer marketing names, like “Core 2” and “Athlon”, while the data suggests that a more precise approach, such as determining individual manufacturer code names, could allow for better and finer-grained processor detection.

We chose to use a 1-nearest neighbor classification scheme due to this marketing name mismatch. For example, a test vector made from samples from both Yonah and Penryn-based Pentium Dual Core CPUs will average to somewhere between the two, leaving each individual Dual Core sample lying closer (and classified as) to the (Yonah) Core and (Penryn) Core 2 test vectors, respectively. The 1-nearest neighbor classification scheme avoids this averaging issue, at the expense of slightly higher noise in the results. With a larger and more precise data set, we posit that the test-vector-generation approach would also work for detecting CPU architectures.

Overall, our JavaScript fingerprinting techniques are able to infer underlying hardware details which were previously unexposed to JavaScript APIs. We believe that more targeted work in this area will allow the inference of processor types with much higher reliability and accuracy.
### Table 1.5. Detected CPU Architecture under Firefox 3.6

<table>
<thead>
<tr>
<th></th>
<th>Atom</th>
<th>Pentium 3</th>
<th>Pentium 4</th>
<th>Pentium D</th>
<th>Pentium M</th>
<th>Core</th>
<th>Pentium Dual Core</th>
<th>Core i3</th>
<th>Core i5</th>
<th>Athlon Classic</th>
<th>Athlon 64</th>
<th>Athlon II</th>
<th>Phenom II</th>
<th>Sempron</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Atom</strong></td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Pentium 3</strong></td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Pentium 4</strong></td>
<td>-</td>
<td>-</td>
<td>40</td>
<td>3</td>
<td>-</td>
<td>3</td>
<td>5</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Pentium D</strong></td>
<td>1</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Pentium M</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Core</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Pentium Dual Core</strong></td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>30</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td><strong>Core 2</strong></td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>41</td>
<td>78</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Core i3</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Core i5</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>7</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Athlon</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Athlon 64</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>-</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Athlon II</strong></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><strong>Phenom II</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td><strong>Sempron</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>
Strangely, in our tests, the total time taken to complete our benchmarks did not scale linearly with clock speed, even given an identical browser, operating system, and processor architecture. Since we expected overall performance to be a fairly good indicator of processor clock speed, these discrepancies may point toward additional sources of system-specific information, which could further enhance the user’s architectural fingerprint.

1.2.4 JavaScript Test Selection

In our fingerprints, we use all 26 tests from SunSpider 0.9 and 9 tests constructed from the V8 Benchmark Suite v5 along with 4 custom tests, for a total of 39 individual benchmarks. The custom tests exercise straight-line code, nested for loops, random number generation, and basic arithmetic. Our fingerprinting techniques, however, use 38 of these tests to perform classification. We chose to discard the “splay” test from V8 due to its excessive completion time (over 2000 ms; other tests might take tens or hundreds of milliseconds). As demonstrated in Figure 1.1, left unchecked, “V8-splay” dominates the normalized vector of test times for both Firefox and Safari, marginalizing any interesting effects we might observe through the other tests. Therefore, we discard the splay test when computing fingerprint vectors (and note that best normalization results are achieved when all test timings are of similar magnitudes).

As previously noted in Section 1.2.3, this particular set of 38 tests performs badly when differentiating between minor versions of Firefox. After experimentation, we discovered that better results were achieved after removing the V8 test “regex” from consideration as well, leaving a 37-element vector for differentiating minor Firefox versions.
1.3 NoScript Whitelist Fingerprinting

NoScript is a Firefox browser plug-in aimed at protecting users against malicious JavaScript on the web. The recommended usage model is to whitelist trusted sites: the NoScript website suggests that JavaScript be blocked by default, and recommends that the user allow scripts “only by trusted web sites of your choice (e.g. your online bank)”. As the web becomes more interactive, allowing JavaScript is becoming increasingly necessary to preserve website functionality. The NoScript whitelist, then, may contain potentially sensitive information about a user’s browsing habits, such as her preferred news sources, banks, webmail provider, and entertainment. Individual whitelist entries can thus reveal sensitive personal information. Just as importantly, the fact that users who visit different sites will allow JavaScript from different sites means that whitelists can be used as additional form of fingerprint.

The NoScript whitelist is very flexible, allowing entries to be either Full Addresses (http://www.noscript.net), Full Domains (www.noscript.net), or Base 2nd Level Domains (noscript.net). By default, NoScript will populate its whitelist with Base 2nd Level Domains, allowing JavaScript for a domain and all of its subdomains. With this in mind, in this chapter we consider all subdomains to be in the same protection domain as their parent. However, note that should a user switch to using either Full Addresses or Full Domains in their whitelist, she will still whitelist addresses which correspond to scripts she wishes to run. The JavaScript allowed by these entries is therefore executable in her browser and can be detected by our fingerprinting methods.

Technically speaking, the NoScript plugin blocks not only the execution of untrusted scripts, but even the fetching of such scripts from the server. Within a trusted JavaScript execution context, an untrusted script simply does not exist. No exceptions are thrown; the contents of the untrusted script are simply missing. As we will show, by
using these facts a malicious website, once allowed to execute any JavaScript at all, can probe the contents of the user’s NoScript whitelist.

Lastly, this attack is only mountable if a NoScript user allows malicious JavaScript to execute. However, many sites fail to display content or work properly without the use of JavaScript. The current best framebusting defense [123] even hides page content by default, requiring JavaScript execution to display anything at all. The attacker could also lure user cooperation via the offer of a game or service, either of which could require JavaScript functionality.

1.3.1 Attack Methodology

We begin by describing how an attacker can determine whether a particular domain is in the user’s NoScript whitelist, and how she can then combine multiple such checks to compute a whitelist fingerprint for the user. Performing these checks requires locating suitable indicator JavaScript scripts on the sites to query; we describe an effective automated spidering approach for finding such scripts. As noted above, we grant our attacker the capabilities provided by the Web attacker or rogue Tor exit node model, as appropriate.

Domain Checking

To create a page which checks for a given domain in a NoScript whitelist:

1. Find a URL in the domain containing JavaScript, suitable for inclusion in a <script> tag.

2. Inspect the JavaScript for the name of a defined object, such as a variable or function.

3. Create a page with two elements:
- A `<script>` tag referencing the script URL.
- JavaScript code checking for the existence of the defined object.

When a user visits this page with JavaScript enabled, one of two things occurs: Either the object exists or is undefined. In the first case, JavaScript is enabled for the tested domain. Otherwise, the domain is blocked (via NoScript or some other means). Note that other browser plugins, such as AdBlock Plus, may block the script’s execution. We ignore the effects of such plugins, as they effectively modify the NoScript blacklist.

Many scripts, when removed from their native locations, throw errors or otherwise fail during this test. Generally, this is due to missing components in the execution environment, such as other JavaScript elements from their domain or an unexpected DOM tree. If such an error occurs, execution of the script stops immediately. However, the functions and variables it defines are still inserted into the global `window` object, and thus our domain test will be successful.

Site Spidering

Any effective fingerprinting solution needs to encompass a significant number of possible domains. Manual creation of a domain-checking page is fairly trivial, but the creation of hundreds or thousands of these pages represents significant human effort. Fortunately, automation of this process is fairly easy.

To produce our domain-checking pages, we built a simple web spider. Given a domain, it simply crawls the first ten in-domain URLs, looking for `<script>` tags. Any scripts included from domains other than the one we are targeting are ignored, as presumably these are not necessary to the operation of the site in question. When an appropriate script URL is found, we execute the code using the V8 JavaScript engine¹

¹Online: http://code.google.com/p/v8/
with an EnvJS² environment. If this execution finds a properly defined object, such as a variable or function, the spider has all it needs to create a NoScript test page for the domain. Figure 1.2 is an automatically-created domain test page for google.com.

Whitelist Fingerprinting

Once a suitable number of domain test pages are created, we can turn our attention to delivering the tests to the machine being fingerprinted and collecting the results into some usable format. To achieve this, we create a test suite from the individual test pages.

The test suite consists of an HTML page and associated JavaScript. To test a given domain, an iframe, containing the test page for the domain to test, is created and inserted into the page. Once the test page reaches a determination, it modifies its location.hash attribute. As each test completes, its iframe is destroyed, releasing its resources and, importantly, stopping all JavaScript execution in the frame. Since each iframe potentially includes arbitrary JavaScript, ceasing its execution as quickly as possible reduces browser load and allows for faster testing.

²Online: http://www.envjs.com/
To prevent serious performance degradation, the test suite limits the number of active tests at once. This prevents a slow-to-respond site from blocking the fingerprinting progress, while not noticeably reducing browser responsiveness. Currently, we simply set this limit to a small constant. Conceivably, this could be automatically adjusted for network lag and browser speed to reduce testing times even further.

This approach does have its limitations. Most notably, the testing process creates continual change in the browser’s chrome. As each iframe loads from the test server, the text on the tab associated with the test flashes between its HTML page title and “Loading...”. Also, test progress is noted in the status bar, with messages such as “Connecting to www.google.com” and “Transferring data from www.google.com”. When NoScript blocks a script, it adds a yellow information bar to the bottom of the browser window. However, once the last iframe is removed, this NoScript information bar disappears as well. These limitations suggest that the optimal way to run the fingerprinting process is in a popunder or otherwise hidden tab, where these notifications will go unnoticed by an unsuspecting user.

Note that the attacker’s site must itself be authorized to execute JavaScript for the fingerprinting to succeed. A Web attacker may be able to trick the user into temporarily adding her site to the whitelist, for example through the promise of a JavaScript-enabled game or other interesting content. A rogue Tor exit node will simply inject the script into a trusted-by-default domain.\(^3\)

### 1.3.2 Prevalence of Testable JavaScript

To measure the effectiveness of our techniques, we attempt to create whitelist probes for each of the Alexa Top 1,000 sites. Starting the spider at the root of each site,

\(^3\)An earlier version of the Torbutton FAQ recommended against the use of NoScript because it would “allow malicious exit nodes to compromise your anonymity via the default whitelist (which they can spoof to inject any script they want).”
we investigate 10 pages in a breadth-first fashion. As described in Section 1.3.1, we stop when we find an appropriate script, which we define as any includable script available in the domain or any subdomain thereof.

Out of the Alexa Top 1,000 sites, we find 706 sites which fit our criteria, and generate NoScript whitelist test scripts for each of these domains.

There are several reasons why a site might not be testable:

1. No JavaScript
2. JavaScript only embedded in HTML pages (no .js files)
3. JavaScript files not accessible within first 10 pages
4. All JavaScript served from a different domain
5. Crawling forbidden by robots.txt

(Note that real attackers would not be constrained by the robots.txt file and could crawl many more pages in each site, looking for smaller scripts that would load and execute more quickly. Our findings are thus a lower bound on the effectiveness of our fingerprinting technique.)

For example, yahoo.com serves all of its external JavaScript files from yimg.com, while facebook.com disallows crawling, requires a login to get past the first few pages, and uses fbcdn.net to serve their JavaScript. Manual intervention may be required in these cases to recognize site-specific content distribution networks (CDNs), or high-value sites that request logins before presenting any interactive content (such as banks or webmail providers).

Once we acquired the set of test scripts, we had to manually remove 17 misbehaving tests. Most of these scripts contained elements such as alert() calls, framebusting
code, attempts to set `document.domain`, or illegal characters. One of these scripts appears to be programmatically generated, and changes every few seconds (rendering our preinspection worthless). Another attempts to fetch a resource from our test server, and throws an error before creating the variable of interest. Lastly, our JavaScript execution engine simply fails on one script, leaving our test page useless. We did not inspect the affected sites for alternative scripts, although issues such as these could potentially be detected and discarded during the site crawl phase.

Therefore, our minimal crawler is able to generate usable test scripts for 68.9% of the Alexa Top 1000 sites. We note again that this is a lower bound, and additional (and less polite) searching would likely reveal suitable scripts at more sites.

### 1.3.3 Fingerprinting Speed

For user fingerprinting systems, speed is paramount. More elements tested means a more unique, and therefore more useful, fingerprint. To determine the speed of probing NoScript whitelists, we created a test suite (as described in Section 1.3.1) to examine all 689 checkable domains we found in the Alexa Top 1000. The test suite creates an iframe for each tested domain and loads a domain test page. This page attempts to fetch a script from the remote site and can then test for success. Once an answer has been determined, the domain’s iframe is deleted, which stops any further script execution. To reduce the impact of high-latency servers, the test suite runs five independent iframes simultaneously.

We tested our benchmarks on Firefox 3.6.11 with NoScript 2.0.3.5. The test machine consisted of an Intel Core 2 Duo processor at 2.66 GHz with 4 GB of RAM. The test suite was served off of a machine on the UC San Diego network.

We ran the test suite twice, once with NoScript configured to block all domains and once with NoScript installed but configured to allow all scripts.
With NoScript disabled, Firefox behaves normally, fetching all requested resources and executing all scripts. We execute the test five times. Each time, we clear all caches and restart Firefox to ensure that all necessary resources were fetched from their remote servers.

The mean time to completely test all 689 domains is 118.6 seconds, with a maximum of 131.9 seconds. Since our benchmark machine and test server are on the same network, fetching each of the domain test pages is a minimal cost. Most of the overall test time is spent fetching and executing the remote scripts. The CDF of domain test times (including iframe creation, test page fetch, remote script fetch, and script execution) across all five runs is shown in Figure 1.3.

With NoScript enabled, Firefox refuses to run all scripts that don’t appear on the NoScript whitelist. Helpfully, NoScript will disable even fetching a blocked script. This removes the network round trip associated with loading a domain’s script (although not the original test page fetch). In accordance with our threat model, we whitelist our test suite, allowing JavaScript to run only for us.
We execute the test suite five times. Each time, we clear all caches and restart Firefox.

The mean time to completely test all 689 domains is 22.2 seconds, with a maximum of 23.3 seconds. Since no external requests are made, most of this time consists of iframe setup, test page fetch, and test suite delays. The CDF of domain test times across all five runs is shown in Figure 1.4.

Notably, as there are no external script fetches, this fingerprinting session runs much faster than before. Also, as none of the sites being tested receive any network traffic from the system being fingerprinted, they are unable to detect fingerprint attempts through log inspection.

Figure 1.4. Completion Time CDF for domain whitelist check (NoScript On)
In Practice

Our fingerprinting methodology expects NoScript to be installed and in use on a user’s browser. As the user repeatedly visits sites of interest which require JavaScript, the user will likely whitelist those sites to streamline their browsing experience. When the user encounters a fingerprinting attempt, then, we expect that most domains are blocked while a select few are allowed. For the allowed domains, we suffer network fetch and script execution time penalties. In this model, then, the total time taken to fingerprint a particular user lies somewhere between the fully-blocked and fully-allowed cases.

In this chapter, we do not attempt to optimize the fingerprinting process. However, there are a few simple ways in which optimizations could be applied. For example, test pages could each test multiple domains, instead of just one. This would cut down on the number of iframes created, along with the number of elements fetched from the test suite server. Care must be taken to eliminate potential interactions between the unrelated scripts, but it is easy to imagine methods of making this possible. If we expect that each whitelist contains relatively few domains, then a less careful approach to testing will give almost as much speedup even when scripts might interact negatively with each other: test several domains in one iframe and, if any of them appear to be allowed, repeat the tests in separate iframes. Moreover, running more simultaneous iframes might allow more domains to be tested in parallel, at the cost of browser set-up time. And, of course, the attacker can test that a user has enabled the NoScript extension before proceeding with the fingerprinting by checking that a script from some other attacker-controlled domain isn’t loaded and run.
1.4 Conclusions

We have described two new ways to fingerprint browsers. The first is based on the relative performance of core Java-Script operations, the second based on probing entries in the domain sitelist of the NoScript Firefox extension. Our new fingerprints can be used by large-scale service providers to harden user accounts against hijacking, whether through phishing or other password compromise. They can be deployed alongside most other forms of browser fingerprint, including cookies and other client-side datastores, the contents of the cache and the history file, and browser and plugin functionality differences. We have developed proof-of-concept implementations of our fingerprinting techniques, showing that our JavaScript performance fingerprinting can distinguish between major browser versions, operating systems and microarchitectures, and that our NoScript whitelist fingerprint can be used to efficiently query almost 70% of the Alexa Top 1000 domains.

Our implementations represent a lower bound on the effectiveness of our techniques. We have shown that it is possible to distinguish between browsers, along with the underlying system hardware and software, based solely on scripting benchmarks. We believe that a finer-grained approach to JavaScript performance fingerprinting can provide even more detailed information, such as hardware revisions within a processor family, clock speed, cache size, and the amount of RAM on the target system. Secondly, extending our technique to mobile devices should produce excellent results, given their unique and constant combination of mobile browser, operating system, and mobile hardware. Also, we believe that NoScript whitelist fingerprinting can be deployed relatively efficiently and against a larger fraction of top sites than we have currently shown.

In future work we hope to deploy a measurement study, modeled after that of Eckersley [43], to measure the effective entropy from our fingerprints. We believe that
there will be sufficient entropy in users’ browsers, hardware configurations, and NoScript whitelists to usefully augment current fingerprinting techniques.

Acknowledgments

We thank Steve Checkoway, Eric Rescorla, Tom Ristenpart, and Stefan Savage for helpful discussions and the W2SP reviewers for their comments.

This material is based upon work supported by the National Science Foundation under Grants No. CNS-0831532 and CNS-0964702, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

Chapter 1, in part, is a reprint of the material as it appears in Web 2.0 Security and Privacy (W2SP) 2011. Mowery, Keaton; Bogenreif, Dillon; Yilek, Scott; Shacham, Hovav, 2009. The dissertation author was a primary investigator and the primary author of this paper.
Chapter 2

Pixel Perfect: Fingerprinting Canvas in HTML5

Tying the browser more closely to operating system functionality and system hardware means that websites have more access to these resources, and that browser behavior varies depending on the behavior of these resources.

We propose a new system fingerprint, inspired by the observation above: render text and WebGL scenes to a `<canvas>` element, then examine the pixels produced. The new fingerprint is consistent, high-entropy, orthogonal to other fingerprints, transparent to the user, and readily obtainable.

2.1 Introduction

Browsers are becoming increasingly sophisticated application platforms, taking on more of the functionality traditionally provided by an operating system. Much of this increasing sophistication is driven by the HTML5 suite of specifications, which make provisions for a programmatic drawing surface (`<canvas>`), three-dimensional graphics (WebGL), a structured client-side datastore, geolocation services, the ability to manipulate browser history and the browser cache, audio and video playback, and more.
The natural way for browsers to implement such features is to draw on the host operating system and hardware. Using the GPU for 3D graphics (and even for 2D graphics compositing\(^1\)) provides substantial performance improvements, as well as battery savings on mobile devices. And using the operating system’s font-rendering code for text means that browsers automatically display text in a way that is optimized for the display and consistent with the user’s expectations.\(^2\)

This chapter proceeds from the following simple observation: Tying the browser more closely to operating system functionality and system hardware means that websites have more access to these resources, and that browser behavior varies depending on the behavior of these resources. The first part of this observation has security implications: codebases not designed to handle adversarial input can now be exposed to it.\(^3\) The second part of the observation, which we focus on, has privacy implications: different behavior can be used to distinguish systems, and thereby fingerprint the people using them.

**Our results**

We exhibit a new system fingerprint based on browser font and WebGL rendering. To obtain this fingerprint, a website renders text and WebGL scenes to a `<canvas>` element, then examines the pixels produced. Different systems produce different output, and therefore different fingerprints. Even very simple tests — such as rendering a single sentence in a widely distributed system font — produce surprising variation. The new fingerprint has several desirable properties:

\(^1\)For example, IE9 uses the GPU for compositing, and recent releases of Chrome use the GPU to accelerate 2D operations on the canvas.

\(^2\)By contrast, the first release of Safari for Windows imported font rendering code from Mac OS X, which offended some users; see [http://www.joelonsoftware.com/items/2007/06/12.html](http://www.joelonsoftware.com/items/2007/06/12.html).

\(^3\)Indeed, one test in the WebGL conformance suite induces a hard system crash on many systems [48]; and the TrueType font handling code in Windows and OS X, which is exposed to attackers by the WebFont specification, was patched to fix an exploitable parsing vulnerability as recently as December of last year [98, 33].
• It is consistent. In our experiments, we obtain pixel-identical results in independent trials from the same user.

• It is high-entropy. In 294 experiments on Amazon’s Mechanical Turk, we observed 116 unique fingerprint values, for a sample entropy of 5.73 bits. This is so even though the user population in our experiments exhibits little variation in browser and OS.

• It is orthogonal to other fingerprints. Our fingerprint measures graphics driver and GPU model, which is independent of other possible fingerprints discussed below.

• It is transparent to the user. Our tests can be performed, offscreen, in a fraction of a second. There is no indication, visual or otherwise, that the user’s system is being fingerprinted.

• It is readily obtainable. Any website that runs JavaScript on the user’s browser can fingerprint its rendering behavior; no access is needed besides what is provided by the usual web attacker model.

Our fingerprint can be used as a black box or as a white box. A website could render tests to a `<canvas>`, extract the resulting pixmap, then use a cryptographic hash to obtain a short, convenient fingerprint. Because the fingerprint is consistent, the pixmap (and therefore its hash) will be identical in multiple runs on one machine, but take on different values depending on hardware and software configuration. This is a black-box use of the fingerprint, since it extracts distinguishing entropy without being concerned with the implementation details.

Alternatively, a website could use a particular test pixmap as evidence that a user is running some particular configuration of browser, operating system, graphics driver, GPU, and, perhaps, display. To identify a user system, the site can compare the pixmap...
it produces against a labeled corpus, such as the corpus we obtained using Mechanical Turk. An intriguing possibility is that GPU quirks could be used to identify a pixmap without comparing against a corpus. However it is performed, such a white-box use of our fingerprint in this way reveals private information about users’ systems.\(^4\) It could also be used to target an attack more precisely, by identifying specific vulnerable system configurations. Trying to exploit only those systems that appear likely to be vulnerable could reduce the number of crashes caused by the attack, and therefore the likelihood that it is detected by the operating system vendor.

Fingerprints on the web have constructive and destructive uses [99]. A use is constructive if users benefit from being fingerprinted. For example, a bank could fingerprint a user’s machine, then require additional authentication for login attempts from systems whose fingerprint does not match. A use is destructive if users do not benefit from being tracked, or do not wish to be tracked. Users can attempt to avoid tracking by using their browsers’ “private browsing” modes [4] or the Tor anonymity service [40].

Users of Tor may be willing to endure a slower, less attractive browsing experience to avoid being tracked. (Note that, although Torbutton disables WebGL, it allows text rendering to a `<canvas>`, and is thus at present partly vulnerable to our fingerprint.) For mainstream browser users, however, the possibility of fingerprinting might be an unavoidable consequence of browsers’ closer ties to operating system functionality and system hardware.

\(^4\)As evidence that such information is private, we note that Chrome knows a great deal about the graphics subsystem—see `chrome://gpu`—but does not expose this information to JavaScript.
Related work: Fingerprints on the web

The earliest mentions known to us of using differences in GPU rendering to fingerprint users are in 2010 discussions on the WebGL mailing list about whether the WebGL renderer information available to JavaScript should provide information about the GPU and driver. Steve Baker argued [10] that it is possible to identify a GPU without this information:

I bet that if I wrote code to read back every glGet result and built up a database of the results - and wrote code to time things like vertex texture performance - then I bet I could identify most hardware fairly accurately.

Benoit Jacob later observed [71] that:

We haven’t yet started accounting for GPU rendering analysis (not just WebGL: in the upcoming generation of browsers, most rendering goes through the GPU and is subject to GPU/driver/config-based rendering differences.

Jacob also suggests the fingerprinting approach we take: “Rendering analysis could proceed by rendering stuff into a canvas 2D and getting its ImageData.” One way to view our research is as demonstrating experimentally that Baker and Jacob were correct in expecting substantial additional leakage from GPU-based rendering. In addition, we show that there is substantial information leakage from font rendering to `<canvas>`.

Many other researchers have proposed techniques to fingerprint web users. These techniques rely on many browser features, including the history and file cache [78], information in HTTP headers and available plugins [94, 43], differences in JavaScript and DOM API support [47], JavaScript performance [99], available fonts [17], and deviations from JavaScript standards conformance [118].
2.2 HTML5 and CSS3

In this section, we introduce the emerging web technologies used in our experiments. First, we present information about the `<canvas>` element, a major portion of what is termed HTML5\(^5\), along with its support for text rendering. Next, we examine WebFonts, part of the CSS3 specification\(^6\). Lastly, we briefly discuss WebGL, an experimental specification\(^7\) currently managed by the Khronos Group (which also maintains the OpenGL specification).

These three specifications are not finalized, and so could change in ways that benefit or hinder fingerprinting success. However, our fingerprinting mechanisms use extremely basic features of these platforms, such as rendering text and inspecting pixels — removal of these features would be dramatic indeed.

2.2.1 HTML5 Canvas

One of the most interesting new elements in HTML5, `<canvas>` provides an area of the screen which can be drawn upon programmatically. It enjoys widespread support, being available in the most recent versions of Chrome, Firefox, Internet Explorer, Opera, and Safari as well as Mobile Safari and Android Browser.

The basic approach to drawing on a canvas is simple: acquire a graphics context, and use the context’s API to effect your changes. In the current HTML5 specification, the only defined context is “2d”. The 2d context provides basic drawing primitives such as `fillRect`, `lineTo`, and `arc`, as well as more complicated features such as Bézier curves, color gradients, and copying in an existing image.

\(^5\)http://www.whatwg.org/specs/web-apps/current-work/
\(^6\)http://www.w3.org/TR/css3-fonts/
\(^7\)http://www.khronos.org/registry/webgl/specs/1.0/
Figure 2.1. JavaScript code to render text on a canvas

Canvas Text

We chose to focus on the text support found in the 2d context. Given a font size, family, and baseline, the 2d context can draw any arbitrary text string to the canvas. No wrapping is performed; the 2d context will happily draw text directly off the edge of the canvas. Lastly, `<canvas>` supports CSS-like text styling, allowing for any combination of font and size. For an example of how text is rendered, see Figure 2.1.

Pixel Extraction

In order for `<canvas>` to be a useful fingerprint, there must be some way to examine its behavior. Fortunately, `<canvas>` makes this extremely easy, providing several ways to inspect its data with pixel accuracy.

First, the 2d context provides the method `getImageData()`. Given a rectangular region of the canvas, this method returns an `ImageData` object. Contained in this object are the RGBA values (as integers) for every pixel in the requested region.

Second, the canvas object itself provides a `toDataURL(type)` method. When passed “image/png”, this method returns a data url consisting of the Base64 encoding of a PNG image containing the entire contents of the canvas. As this is a very convenient canvas-level method, we used this approach to extract data in our experiments. During black-box use of these fingerprints, the test suite could simply hash these data URLs, thereby removing the need to upload entire images from each client.
It is worthwhile to note that these methods do preserve the same origin policy — if an image from a different origin has been drawn on this canvas, they will throw a `SecurityError` exception instead of returning pixel data. Therefore, our `<canvas>` fingerprints must only contain image resources that are under our control.

### 2.2.2 WebFonts

WebFonts, specified in CSS3, allow web designers to load a font face on-demand, rather than relying solely on the fonts installed on each client machine. To include a font, the web designer inserts a `@font-face` CSS rule with a `src` attribute pointing to a font in an appropriate format. The browser then downloads the font and makes it available for use on the page. Fortunately for us, web fonts can be used when writing to a `<canvas>`.

To include WebFonts, we depend on the WebFont Loader, co-developed by Google and Typekit. With this library, WebFonts can be loaded solely through the use of JavaScript, and callbacks can be established for certain events (such as the font becoming available or, conversely, failing to load). By attaching our rendering to a successful load, we are guaranteed to use the correct font while writing to the canvas.

### 2.2.3 WebGL

WebGL provides a JavaScript API for rendering 3D graphics in a `<canvas>` element. Modeled after OpenGL ES 2.0, WebGL is currently a draft specification and implemented and enabled in Chrome, Firefox, and Opera, as well as implemented but disabled in Safari. Each of these browsers provides a hardware-accelerated implementation, using the installed graphics hardware to render each frame. To mitigate serious misbehaviour and crashes, all of these browsers enable WebGL only for a whitelisted set of graphics cards and drivers.

---

8[https://developers.google.com/webfonts/docs/webfont_loader](https://developers.google.com/webfonts/docs/webfont_loader)
Current WebGL implementations expose their functionality through a separate canvas context (which will eventually be named “webgl”). The WebGL API is too complex to describe here in sufficient detail, but is stylistically similar to the desktop OpenGL API. It provides for vertex and fragment shaders, written in OpenGL Shading Language (GLSL), that, after compilation, run directly on the graphics card. WebGL also provides for OpenGL-style textures, as well as different lighting primitives. More advanced techniques, such as specular highlighting, bump mapping, and transparency, can be achieved through custom GLSL shaders.

2.2.4 Security Implications

These new capabilities, while providing more and more ways for developers to produce interesting and useful web content, do come at a cost. For efficiency’s sake, inputs from the web are passed farther and farther down the software stack: for example, GLSL shaders are compiled directly from web pages and run on the graphics card, allowing arbitrary data to pass between the JavaScript execution engine and the kernel-level graphics driver. Other attack surfaces are possible: malicious or misguided GLSL shaders can crash or hang the entire operating system on OSX and Windows XP or cause GPU resets on Windows 7 [48].

WebFonts, while appearing more innocent, can also be a security concern. Remote code execution vulnerabilities while parsing TrueType fonts have been discovered in Windows [98], OSX, Debian, Red Hat, and iOS [33].

While we do not use these exploits in this chapter, we take advantage of the fact that these new web technologies, for efficiency’s sake, push untrusted web content deep into the operating system stack. In our case, however, we simply examine the results of these operations, exposing differences in implementation (however slight).
2.3 Experiments

In this section, we discuss the tests that underly our fingerprinting scheme, as well as the support infrastructure we built in order to deliver the tests and inspect their results. We will also detail the process of fingerprint collection from a large number of disparate users on the web.

2.3.1 Tests

For our fingerprints, we use six tests: text_rial, text_rial_px, text_webfont, text_webfont_px, text_nonsense, and webgl. Each test follows the same basic outline: render test data to a canvas and extract its contents as an encoded PNG.

Arial Text

In our first two tests, we render a short sentence in Arial, a font known for its ubiquity on the web. To exercise each letterform, we use the pangram “How quickly daft jumping zebras vex.,” along with some added punctuation.

For text_rial, the text is rendered to the canvas in 18pt Arial. In text_rial_px, we change the font specification to 20px Arial. Example images produced by these two tests are shown in Figure 2.2. The actual code for these two tests is almost identical to the snippet in Figure 2.1 — complicated tests aren’t needed for fingerprinting!
These two tests are extremely similar to the Arial tests, with the added complexity of loading a new font from a web server. In a more sophisticated or targeted fingerprint, the delivered font could be carefully tuned by the fingerprinter to exercise corner cases in font loading.

In our case, however, we use the WebFont Loader to load “Sirin Stencil” from the Google Web Fonts server\(^9\). Once it loads, we render the same pangram as in our Arial tests. For text_webfont, the text is set in 12pt Sirin Stencil, while text_webfont_px uses 15px Sirin Stencil. Example images are shown in Figure 2.3.

Nonsense Text

Code-wise, this test is nearly identical to the two Arial tests. However, instead of a valid font specification, we set the 2d font specification to “not even a font spec in the slightest”. This exercises the fallback handling mechanisms in the browser: what does it do with an invalid font request? The browser’s choice of fallback font, as well as its positioning and spacing, can be quite telling.

Also, note that this behavior is also the fallback font handling mechanism for when the browser is presented with a valid font specification for an unavailable font. Using this technique, tests can be written to probe for the existence of a particular font on target machines. If enough of these tests are run, the fingerprinter can derive a fairly comprehensive list of the installed fonts on the target machine.

\(^9\)http://www.google.com/webfonts
An example output is shown in Figure 2.4.

WebGL

`webgl` is our only test whose code spans more than a few lines. As WebGL scenes go, however, this scene is almost minimal. We create 200 polygons, approximating the hyperbolic paraboloid \( z = \frac{y^2}{2} - \frac{x^2}{3} \), with \(-3 \leq y \leq 3\) and \(-3 \leq x \leq 3\). Over this surface, we apply a single texture: a 512 by 512 pixel rasterized version of ISO 12233, the ISO standard for measuring lens resolution. Designed for measuring sharpness and resolution in electronic still-picture cameras, this texture contains many areas with high detail. We then add an ambient light with color \((0.1, 0.1, 0.1)\) and a directional light of color \((0.8, 0.8, 0)\) and direction \((2,4,9)\). Placing our surface at \( z = -10 \), we render this simple tableau.

A example, rendered on OS X 10.7.3 with Chrome 18 on a AMD Radeon HD 6490M, is shown in Figure 2.5.

Test Speed

Speed is an important characteristic for fingerprints. Tests that take minutes are categorically less useful than tests that take seconds, especially if they are deployed to protect online accounts (imagine waiting even twenty seconds to log into your webmail!). In our case, each test takes a mere fraction of a second to run — indeed, the longest delays occur while fetching the image assets. Consisting of a 76 KiB PNG image and a 24 KiB WOFF font, these fetches would not be out of place during any page load on today’s web. For comparison, once it has loaded, the Quake 3 WebGL Demo\(^\text{10}\) runs at

\(^{10}\text{Quake 3 Demo: http://media.tojicode.com/q3bsp/} \)
Figure 2.5. An example of WebGL

60 FPS with around 15% CPU utilization in Chrome 18 with a 2 GHz Core i7 and an AMD Radeon HD 6490M. There is room for a substantial number of <canvas>-based fingerprints to run before adversely affecting user experience.

2.3.2 Infrastructure

In general, web designers can depend on their sites rendering in a consistent manner across various browsers and operating systems. Therefore, we expected that any fingerprintable differences will be subtle, perhaps not even visible to a human observer.

To view these trace differences, we built a small webapp which can administer the tests and examine their results. Experiments are served as pure JavaScript, and results are collected as data URL-encoded PNGs. Our framework then compares these results as images, allowing it to group identical results and display pixel-level differences between these groups.

We use two types of image comparison: pixel-level difference and difference maps. When constructing a pixel-level difference, the framework first creates a new image
of the appropriate size. Then, each pixel’s color is set to the channel-wise difference between the two images at that location. If this color is anything other than transparent pure black (which indicates that there is no difference between the two images at this pixel), we set the alpha value of the differing pixel to 255, rendering it fully opaque. For difference maps, each pixel in the map is set to either white or black, depending on whether the original images differ. A purely white difference map indicates identical images, while perfectly black indicates difference in every pixel.

Unfortunately, we do not have ready access to several hundred distinct consumer-level computer systems. To collect enough data to demonstrate the applicability of our fingerprints, we turned to Amazon Mechanical Turk. We modified our framework to deliver multiple tests on a single page, along with extended instructions for the human workers on Mechanical Turk. The next section explains our process and results more thoroughly.

2.3.3 Data Collection

We collected samples from 300 distinct members of the Mechanical Turk marketplace, paying each a small sum to report their graphics card and graphics driver version. Meanwhile, our five fingerprinting tests ran in the background. We also collected various metadata, such as the browser’s user agent string and the WebGL-reported renderer, vendor, and version.

In Safari, acquiring information on the user’s graphics card and driver is trivial: simply ask the WebGL canvas context. For example, given a WebGL context gl, calling gl.getParameter(gl.RENDERER) on Safari 5.1.3 might return “AMD Radeon HD 6490M OpenGL Engine”, while calling gl.getParameter(gl.VERSION) might return “WebGL 1.0 (2.1 ATI-7.18.11)”, which includes the version number of the current graphics driver. Currently, however, WebGL is disabled by default in Safari, and must be manually enabled through the Developer menu.
In all three browsers which ship with WebGL enabled, however, these values are redacted. For example, in Chrome 18.0.1025.39, asking for the WebGL renderer returns “WebKit WebGL”, while the version is “WebGL 1.0 (OpenGL ES 2.0 Chromium)”. Firefox and Opera Next are similarly unhelpful, returning generic statements about the version of WebGL, without reference to the installed hardware. We conclude that browser vendors consider hardware and driver version to be identifying information, and any information our fingerprints extract about them can be considered a loss of privacy.

Owing to these constraints, the user-facing portion of our survey asked users of Chrome, Firefox, and Opera to manually report their graphics card and driver. For Chrome and Firefox, the user was instructed to copy text from the browser pages chrome://gpu and about:support, respectively. Opera does not appear to have any such mechanism for hardware discovery, and so these users were asked to discover the information through other means.

Platform Representation

Windows represents the lion’s share of user platforms, with 276 samples. A full 226 of these are from Windows 7, with 9 Windows Vista, 40 Windows XP, and 1 Windows 8 rounding out the total. We also have 13 samples from OS X, ranging from 10.5.8 to 10.7.3, and 11 samples from Linux.

Chrome is overwhelmingly present in our data set, with 222 samples ranging from version 10 to 19. Firefox comprises almost all of the rest, with 71 samples between versions 8 and 11. We also have 4 samples from Opera 9.8, 2 from Safari 5, and 1 from Android.

These numbers do not represent the current software usage of the internet as a whole. We attribute this to text in the survey, asking users to use a WebGL-enabled browser to complete the task. Notably, since Internet Explorer does not support WebGL,
we posit that every Mechanical Turk user using IE either skipped our survey, or returned in either Chrome or Firefox. The overwhelming prevalence of Windows 7 is also correlated with WebGL capability — users who have up-to-date browsers might also be more likely to have up-to-date operating systems.

**Persistence**

During the course of data collection, twenty-three distinct users performed our survey twice. Due to our setup, we collected results for text\_arial, text\_webfont, text\_nonsense, and webgl when they first performed the survey. However, for all twenty-three users, each of these tests were identical to their later submissions. This suggests that our fingerprints are consistent — for a given browser/OS/graphics card platform, performing the same test will always give identical results.

To further test this hypothesis, we ran all of our tests on 5 identically-provisioned lab computers, running Firefox 11 on Windows XP. As expected, all five computers produced identical results on each test, providing further evidence that our fingerprints are stable across identical hardware and software stacks. Additionally, we note that our fingerprints are unable to distinguish between users who use the exact same hardware and software.

**Errors**

While running the experiment, six users experienced failures in the text\-_webfont and text\_webfont\_px tests, returning a blank PNG instead of one containing text. Upon investigation, we attribute these failures to a known race condition in the WebFont Loader library. Since the race condition is a transient error, we ignore these six samples whenever they affect our fingerprint information leakage, as including them will improve differentiation.
Data Quality

We treat the user’s report and user agent string as ground truth. While examining the data, we saw no evidence of any forgery, either of user agent strings or graphics card information.

Since we conducted the survey on Mechanical Turk, however, some level of imperfection is to be expected. Indeed, twenty-four users did not submit their graphics card information, either by copying unrelated text into the survey box or failing to fill in the box at all. In the authors’ favorite submission, the worker simply entered “It was very nice.” into the text field. These unclassified samples are included in our results, since they represent the state of a <canvas> fingerprint in the wild, with no inside information about hardware.

2.4 Results

The most important feature of any fingerprint is differentiation: if every system fingerprinted performs identically, what use is the fingerprint? With this goal in mind, we now examine the results of our tests as applied to Mechanical Turk users.

2.4.1 Arial Font Rendering

In general, we find a surprising amount of differentiation in the ways that fonts are rendered. Even Arial, a font which is 30 years old, renders in new and interesting ways depending on the underlying operating system and browser. In the 300 samples collected for the text_arial test, there are 50 distinct renderings. For text_arial_px, this number drops to 43. This amount of diversity is astounding, showing differences even between computers running the same operating system and browser version. In Figure 2.6, you can see some of the different results that appear. Interestingly, some of the Linux samples shown are not using Arial at all, but are substituting an unidentified yet similar
Font substitutions such as these provide an extra dimension of distinguishability between otherwise matching computers.

The largest cluster of identical renderings on `text_arial` contains 172 samples, taken in versions of Chrome ranging from 15 to 18 on Windows XP, Vista, or 7. We attribute the size of this group to the relative popularity of this browser/OS combination in our data set. However, seven other groups contain samples taken on this platform as well, indicating that other, more hidden variables might be discoverable in this test.

**Classification**

While a repeatable, trivial fingerprint borne out of simple text rendering is quite promising, even more information can be derived with this test. 10 of 50 groups contain samples taken on Linux machines, while 5 contain samples from OS X. Each of the fifty groups contains samples from only a single operating system family, implying that this fingerprint alone is sufficient to distinguish operating systems!

Similarly, almost every group contains samples from only a single browser family. The single exception occurs on OS X, where Chrome and Safari can produce identical renderings. Otherwise, the font handling in each browser is distinctive enough to leave a usable fingerprint.

Given these results, we conclude that rendering a simple pangram in Arial on a `<canvas>` is enough to leak the user’s operating system family and (almost always) browser family.

**Differences**

Collecting text samples from such a wide variety of sources allows us to find slight differences among rendering engines, which could be exploited for better and more precise fingerprints. In Figure 2.7, we present the original rendering for the 172-member Windows/Chrome group, along with several difference maps. The most obvious
### Figure 2.6. 13 ways to render 20px Arial

<table>
<thead>
<tr>
<th>Windows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS X:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, pur</td>
</tr>
</tbody>
</table>

difference occurs across platforms, where we see a marked difference in the kerning of the text, leading to the text rendering at two different lengths. For samples on the same platform, the difference map shows the outline of each letter, indicating that antialiasing or subpixel hinting (such as ClearType) was used. From this, a fingerprinter might be able to deduce information regarding the user’s display or ClearType settings. Finally, the most interesting difference map comes from a single sample, claiming to be Chrome 17.0.963.56 on Windows 8. Here, we see only a few pixels’ difference, located solely near round edges, indicating subtle differences in the font rendering engine in this system. Such differences might be found in currently deployed systems by using other fonts and glyphs, rendering this sort of fingerprint even more potent.

More generally, our technique is capable enough to flush out any differences between two font handling stacks. During our experiments, we observed that at least operating system, browser version, graphics card, installed fonts, subpixel hinting, and antialiasing all play a part in generating the final user-visible bitmap. Additionally, our fingerprint can recognize the impact of any other variables that impact font rendering, such as the exact placement of pixels on a physical LCD screen (which ClearType might take into account). More data and better platform characterization is necessary for identification and individual classification of these more subtle variables.

Lastly, in both Figure 2.7 and Figure 2.6, there is a surprising amount of diversity in the length of rendered text. This suggests an even simpler fingerprinting mechanism: create specially crafted sentences as DOM elements and measure their length via JavaScript. When implemented as a proof-of-concept, this technique shows a measurable and repeatable difference in the length of text between Firefox 10 and Chrome 18 on OSX. This simpler fingerprint should reveal a strict subset of the information that our <canvas>-based text rendering does, but does so using far simpler methods (and thus will be much more difficult to defend against).
Figure 2.7. Difference maps for a group on text_arial
**Entropy**

Fingerprints are useful only inasmuch as they differentiate users. Since our fingerprints reveal differences in hardware and software stacks, popular hardware configurations reduce the identifying power of the tests. Conversely, however, some setups might produce unique results, identifying their user precisely. To estimate the tests’ overall effectiveness, we will compute the distribution entropy of our groupings. This metric indicates how many predictive bits the test reveals, or, more precisely, the differential loss of user privacy resulting from the test. However, since we do not believe that the hardware and software in our 300 samples is a representative sample of the internet as a whole, these metrics should be treated as rough guidelines at best.

To measure distribution entropy, we use the formula

\[
E = - \sum_{i=1}^{n} p(x_i) \log_2 p(x_i)
\]

(2.1)

where \( p(x_i) \) is the size of the \( i \)th group divided by the number of samples. text_arial, across 50 groups, shows a distribution entropy of 3.05 bits. text_arial_px, having only 43 distinct groups, reveals 2.86 bits. We note that these numbers do not represent the true entropy of these tests, as we do not have enough data to accurately model the distribution of platforms in the wild.

### 2.4.2 WebFont Rendering

Using WebFonts allows us to standardize the font under test. Unlike in the Arial tests, where slight differences in the installed font could conceivably exist, any differences in the WebFont test must be a direct consequence of the font engine used to render the text.
In general, `text_webfont` shows a similar distribution to the Arial tests. From
the 294 properly completed tests, there are 45 distinct ways to render our sample sentence.
10 of these are presented in Figure 2.8. As in the Arial tests, each group consists of a
single OS family/browser family pair, with the sole exception of Safari and Chrome on
OS X, lending further evidence that text rendering can uniquely identify platform details.

Interestingly, as is visible in the last Windows sample, some clients did not use
Sirin Stencil at all; rather, they substituted in a font of their own. In our data, we found
five of these samples: four (Chrome 16 and 17, Firefox 10 and 11) using Times New
Roman, the fifth (Opera 9.8) using what appears to be Arial. Samples containing the
correct webfont exist for each of these browsers, so we must assume that these five users
have disabled WebFonts for security reasons, or that there was an error loading the font.
If the former, these users may be doing themselves a disservice: their browsers stand out
quite strongly in our fingerprints. Indeed, three of these samples are unique, with only
the Chrome pair sending identical results.

`text_webfont_px` shows almost identical results, with 44 groups from 294
samples. Therefore, we shall focus only on `text_webfont` from now on.

As in our Arial tests, the largest group consists solely of Chrome on Windows,
with 164 identical samples. Oddly, the Windows 8 sample appears in this group, along
with many of the samples from the large Arial group. This suggests that the font handling
in Windows 8 has not been entirely changed, and similar subtle differences might exist in
other, currently indistinguishable font engines.

Another interesting render is shown in Figure 2.9. 13 separate users submitted
this result, each using Chrome (17, 18) on Windows (XP, Vista, 7). After being surprised
at the relative poor quality of this text, we were able to reproduce it exactly by disabling
the option “Smooth edges of screen fonts” in the Windows 7 Performance Options
preferences, which disables Microsoft’s ClearType subpixel hinting, as well as any
**Figure 2.8.** 10 ways to render 12pt Sirin Stencil

<table>
<thead>
<tr>
<th>Windows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS X:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
<tr>
<td>How quickly daft jumping zebras vex. (Also, punctuation: &amp;/c.)</td>
</tr>
</tbody>
</table>
antialiasing on fonts system-wide. Our text_webfont test, therefore, leaks this setting to an online fingerprinter, and this leakage suggests that other ClearType configuration might be detectable as well. For example, if ClearType performs differently with distinct screen DPIs or LCD pixel layouts, these subtle differences will reveal themselves to this very simple fingerprinting technique, leaking even more information about the user’s hardware.

**Entropy**

text_webfont, with 294 samples in 44 groups, shows a distribution entropy of 2.93 bits. text_webfont_px is similar, at 2.95 bits.

### 2.4.3 WebGL

When we first started this project, we predicted that we would need to try quite a few underhanded tricks in order to see differences between graphics cards. Surprisingly, this is not the case. Our experiments show that graphics cards leave a detectable fingerprint while rendering even the simplest scenes.

As described in Section 2.3.1, our WebGL test creates a single surface, comprised of 200 polygons. It applies a single black and white texture to this surface, and uses simple ambient and directional lights. We also enable antialiasing.

Of the 300 users who participated in our study, 30 submitted no data for this test. In our framework, this absence indicates either that WebGL is disabled or that an error occurred during the test.

Under visual inspection, the 270 remaining images appear identical. When examined at the level of individual pixels, however, we discovered 50 distinct renders.
This level of heterogeneity is, frankly, quite surprising. Our scene is rendered with basic matrix operations, and we expected far more consistency among graphics cards. One possible explanation suggests that graphics cards, in the name of efficiency, cut corners with respect to graphics processing. Perhaps renders are nondeterministic, but in such minor ways as to be undetectable for humans.

Looking at the subset of our data in Table 2.1, we see that this is not the case: most graphics cards produce pixel-perfect output as compared to others of their model. There is also a resemblance among the members of the same line (note groups 1, 5, and 24), suggesting that graphics card manufacturers perhaps share hardware or driver implementations between coexisting and evolving product lines.

The graphics cards, browsers, and operating systems present in all 51 groups can be found in Section 2.7.

**Classification**

Of course, simply knowing that two implementations differ is useful, but understanding how might give clues as to why. In Figure 2.10, we see the original image for group 24, our largest, as well as its difference maps against several other renders.

Examining these in detail, there are several different ways in which these groups differ. In the group 1 and group 36 difference maps, we see that most of the difference is
Table 2.1. Selected groupings of identical WebGL renders. Each group corresponds to a single pixmap.

<table>
<thead>
<tr>
<th>Group</th>
<th>#</th>
<th>Graphics Cards</th>
<th>Browsers</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>ATI Mobility Radeon HD 4250 (Device 9712)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Radeon HD 2400 Pro (Device 94c1)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Radeon HD 2600 Pro (Device 9589)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>ATI Radeon HD 3280 Graphics (Device 9652)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Radeon HD 3800 Series (Device 9505)</td>
<td>Chrome 19</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ATI Radeon HD 4200 (Device 9710)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>ATI Mobility Radeon HD 4300 Series (Device 9552)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Mobility Radeon HD 4330 (Device 9552)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Mobility Radeon HD 4530 (Device 9553)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Mobility Radeon HD 4670 (Device 9488)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Mobility Radeon HD 545v (Device 9555)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Mobility Radeon HD 550v (Device 9480)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Radeon HD 4350 (Device 954f)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ATI Radeon HD 4550 (Device 9540)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>Intel(R) 82945G Express Chipset Family (Device 2772)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) 82945G Express Chipset Family (Device 2772)</td>
<td>Chrome 17</td>
<td>Windows Vista</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) 82945G Express Chipset Family (Device 2772)</td>
<td>Chrome 18</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) 82945G Express Chipset Family (Device 2772)</td>
<td>Firefox 10</td>
<td>Windows XP</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) 82945G Express Chipset Family (Device 2772)</td>
<td>Firefox 8</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics Family (Device 0102)</td>
<td>Firefox 10</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Mobile Intel(R) 945 Express Chipset Family (Device 27a2)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>Intel(R) G33/G31 Express Chipset Family (Device 29c2)</td>
<td>Firefox 10</td>
<td>Windows 7</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>Intel(R) G41 Express Chipset (Device 2e32)</td>
<td>Chrome 15</td>
<td>Windows 7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Intel(R) G41 Express Chipset (Device 2e32)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Intel(R) G41 Express Chipset (Device 2e32)</td>
<td>Chrome 19</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) Graphics Media Accelerator HD (Device 0046)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0042)</td>
<td>Chrome 15</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0042)</td>
<td>Chrome 16</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0042)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Chrome 16</td>
<td>Windows 7</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Chrome 17</td>
<td>Windows 8</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 10</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 10</td>
<td>Windows XP</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 11</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 11</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics (Device 0046)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>Intel(R) HD Graphics Family (Device 0102)</td>
<td>Firefox 10</td>
<td>Windows XP</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Intel(R) HD Graphics Family (Device 0116)</td>
<td>Firefox 8</td>
<td>Windows 7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Intel(R) HD Graphics Family (Device 0116)</td>
<td>Firefox 9</td>
<td>Windows 7</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>NVIDIA GeForce 6150SE nForce 430 (Device 03d0)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 6200 (Device 0221)</td>
<td>Firefox 10</td>
<td>Windows XP</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 7100 / NVIDIA nForce 630A (Device 07e1)</td>
<td>Firefox 10</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 7150M / nForce 630M (Device 0731)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 7300 SE/7200 GS (Device 01d3)</td>
<td>Chrome 17</td>
<td>Windows 7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNKNOWN</td>
<td>Chrome 15</td>
<td>Windows XP</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>NVIDIA GeForce 8600</td>
<td>Chrome 17</td>
<td>Linux</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 9400</td>
<td>Chrome 17</td>
<td>OSX 10.7.3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 9800</td>
<td>Chrome 17</td>
<td>Linux</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NVIDIA GeForce 320M (Device 08a0)</td>
<td>Firefox 10</td>
<td>OSX 10.6</td>
</tr>
</tbody>
</table>
located at the edges of color regions and polygons. This suggests that these graphics cards are performing antialiasing slightly differently, or perhaps simply linearly interpolating textures in almost imperceptibly different ways. In contrast, we see that the renderers in Group 20 produced slightly different colors when lighting the white portions of the texture, as compared to group 24. However, group 23 differs at almost every single non-background pixel. The two renders are visually indistinguishable, suggesting that the differences occur in the very least significant bits of each color.

The most interesting difference, however, appears between group 24 and group 25. These renders differ by only a few pixels! Indeed, the hardware used to produce these images is extremely similar: group 24 consists mainly of Intel G41 (device ID 0x2e32), Intel HD Graphics (device IDs 0x0042, 0x0046, 0x0df1), and Mobile Intel 4 Series (device ID 0x2a42), while group 25 consists solely of Intel HD Graphics systems (device IDs 0x0102 and 0x0116). If we assume that distinct device ID numbers indicate distinct products, these few pixels strongly suggest that even extremely similar graphics systems can be differentiated simply through rendering the right images!

Also, it is worth noting that out of 50 total groups, 49 contain samples from a single operating system family. Group 42 (in Table 2.1) is the only exception, containing both Linux and OS X. However, given the relative lack of samples from Linux and OS X, we cannot determine whether this delineation is due to significant differences in each operating system’s graphics handling, or whether we simply do not have enough samples comparing identical graphics cards across diverse software stacks.

**Entropy**

Due to the large number of unique graphics cards and their consistent effects, WebGL gives a distribution entropy of 4.30 bits, over 300 samples in 51 groups. Again, the actual entropy revealed by this test depends upon the frequency of each graphics card
as deployed in the wild, which we can not extrapolate from our data. Therefore, this entropy should be considered a rough estimate, at best.

2.4.4 Comprehensive Fingerprinting

Given the relative success of each individual test, we shall now examine their efficacy when combined. If their predictive power lies solely in browser and operating system family, we will expect a relatively similar number of distinct groups once all fingerprints are combined. With this approach, each group consists of samples for which all six tests are identical.

Among the 294 samples which successfully completed all six tests, there are 116 distinct groups. The largest of these contains 51 samples, and consists almost solely of Chrome 17.0.963.56 on Windows 7, with a single Windows Vista mixed in. As for graphics cards, it contains Intel G41 Express Chipsets (DID 0x2e32), Intel Graphics Media Accelerator HD 0x0046), Intel HD Graphics (DID 0x0042, 0x0046, 0xdf1), Intel 4 Series (DID 0x2a42), and Intel 45 Express (DID 0x2a42). As mentioned in Section 2.4.3, further and more sophisticated WebGL fingerprints may be able to differentiate these graphics systems.

Entropy

Overall, our fingerprints do combine beneficially: among the 116 groups, our five extremely simple tests show a distribution entropy of 5.73 bits. We believe that more specialized and targeted tests could reveal even more, perhaps down to the exact installed graphics card, operating system, and browser family.
2.5 Defenses

In this section, we propose several methods of preventing \texttt{<canvas>}-based fingerprinting and consider their impact.

First, browser vendors could completely disable canvas pixel extraction. While obviously preventing any potential \texttt{<canvas>}-based fingerprinting, this fix removes a useful capability of the platform—imagine building a webapp for photo editing or drawing. Therefore, let us only consider defenses that do not overly undermine the potential of \texttt{<canvas>}.

One might imagine a defense whereby the browser adds random pixel noise whenever pixels are extracted. Under this regime, directly comparing image results becomes far more difficult. However, slight noise can be easily circumvented: simply repeat the test a few times and compare the results (perhaps by averaging or selecting the most common pixel color). While increasing the noise combats this, it also degrades the performance of \texttt{<canvas>} significantly for legitimate applications. Applying the same noise on multiple runs would only aid in fingerprinting. Therefore, we conclude that adding noise is not a feasible defense against our fingerprinting methods.

Thinking further, we note that our fingerprints measure functional differences in both the hardware and software running on the device. A sure-fire way to defend against leaking this information, then, is for every system to produce identical results. To do so, browser vendors will need to agree on a list of \texttt{"<canvas>-safe"} fonts, and then ship these fonts, along with text rendering libraries such as Pango, as a supplement to the browser. To support WebGL, browsers would ignore the graphics card entirely and render scenes in a generic software renderer such as Mesa 3D. While this approach might be acceptable where privacy is of the utmost importance, the performance impact would be unacceptable in a shipping browser. Note that performing this
emulation is a signal in and of itself, revealing that the user is taking precautions to be anonymous.

The easiest effective defense, then, is to simply require user approval whenever a script requests pixel data. Modern browsers already implement this type of security — for example, user approval is required for the HTML5 geolocation APIs. This approach continues the existing functionality of `<canvas>` while disallowing illegitimate uses, at the cost of yet another user-facing permissions dialog.

More complicated defense schemes can certainly be imagined. Imagine, perhaps, a `<canvas>` implementation that uses hardware acceleration to produce the pixels displayed to the user, but regenerates the entire image with an emulated implementation whenever the site requests pixel data. Such complicated schemes might successfully defend against a `<canvas>` fingerprint, but add significant complexity to HTML5 APIs and behavior.

### 2.6 Conclusions

We have demonstrated that the behavior of `<canvas>` text and WebGL scene rendering on modern browsers forms a new system fingerprint. The new fingerprint is consistent, high-entropy, orthogonal to other fingerprints, transparent to the user, and readily obtainable.

We believe that such fingerprints are inherent when the browser is — for performance and consistency — tied closely to operating system functionality and system hardware.

We do not yet have the data necessary to estimate the entropy of our fingerprint over the entire population of the web, but given our preliminary findings, 10 bits is a (possibly very) conservative estimate. Indeed, Benoit Jacob has estimated the entropy of just the GPU model at 9 bits [70].
We were surprised at the amount of variability we observed in even very simple tests, such as rendering a sentence in 12-point Arial. We conjecture that it is possible to distinguish even systems for which we obtained identical fingerprints, by rendering complicated scenes that come closer to stressing the underlying hardware. Note that an attacker could refine a fingerprint in a black-box way by having some victims render experimental scenes in addition to the ones used for fingerprinting. Those scenes that allow otherwise identical systems to be distinguished should be added to the fingerprint.

We are pessimistic about the possibility of eliminating the fingerprints we identified without seriously degrading browser functionality and performance, or require yet more user approval dialogs to enable basic functionality. Perhaps the time has come to acknowledge that fingerprints are unavoidable on the modern web.

For browsers specifically designed to limit the ability of attackers to fingerprint users, such as Tor’s modified Firefox with Torbutton [110], more drastic steps may be necessary. Torbutton already disables WebGL, and it likely should disable GPU-based compositing and 2D canvas acceleration. But even this step does not address the fingerprint obtained through font rendering. It may be necessary for Tor’s Firefox to eschew the system font-rendering stack, and instead implement its own—based, for example, on GTK+’s Pango library.

Acknowledgments

We are grateful to Úlfar Erlingsson, Eric Rescorla, Stefan Savage, and Geoff Voelker for helpful discussions about this work. This material is based upon work supported by the National Science Foundation under Grants No. CNS-0831532 and CNS-0964702, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

---

11WebGL could perhaps be implemented in software. As an example, Chrome 17 falls back on a software GL backend on systems without a supported GPU.
Chapter 2, in part, is a reprint of the material as it appears in Web 2.0 Security and Privacy (W2SP) 2012. Mowery, Keaton; Shacham, Hovav, 2012. The dissertation author was a primary investigator and the primary author of this paper.

### 2.7 Data Characterization

Table 2.2 is a full characterization of our webgl data set, including the hardware and operating systems which generate each unique result.
Table 2.2. Groups of identical WebGL renders. Number in parentheses indicate device IDs, where appropriate.

<table>
<thead>
<tr>
<th>#</th>
<th>Size</th>
<th>Graphics Cards</th>
<th>Browsers</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>ATI Mobility Radeon HD 4250; ATI Radeon HD 2400 Pro, 2600 Pro, 3200, 3800, 4200</td>
<td>Chrome (17, 19)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>AMD Radeon HD 6250, 6300, 6310, 6670, 6800; ATI Radeon HD 5450, 5670</td>
<td>Chrome (17, 18)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>AMD Radeon HD 6250; Mobility Radeon HD 5650; ATI Radeon HD 5670; Unknown</td>
<td>Firefox (9)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>ATI HD 5850</td>
<td>Opera (9.8)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>ATI Mobility Radeon HD 4300, 4330, 4530, 4670, 545v, 550v; ATI Radeon HD 4350, 4500</td>
<td>Chrome (17)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>ATI Mobility Radeon HD 5470, 5730, 5700 Series; Intel HD Graphics (0116)</td>
<td>Chrome (17)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>ATI Radeon 2100</td>
<td>Chrome (17)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>ATI Radeon HD 2600 Pro</td>
<td>Chrome (17)</td>
<td>OS X 10.6.8</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>ATI Radeon HD 4300/4500 Series (954f)</td>
<td>Firefox (8)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>ATI Radeon HD 5700 Series (68be), 5800 Series (6899); Unknown</td>
<td>Firefox (10, 11)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>ATI Radeon HD 6750M</td>
<td>Firefox (10)</td>
<td>OS X 10.7</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>Unknown</td>
<td>Android 2.3.4</td>
<td>Android</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>Intel GMA 950</td>
<td>Chrome (18)</td>
<td>OS X 10.7.3</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>Intel GMA X3100</td>
<td>Chrome (17)</td>
<td>OS X 10.5.8</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>Intel GMA 950</td>
<td>Firefox (10)</td>
<td>OS X 10.6</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>Intel HD Graphics 3000</td>
<td>Firefox (10)</td>
<td>OS X 10.7</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>Intel HD Graphics 3000</td>
<td>Firefox (12)</td>
<td>OS X 10.7</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>Intel 946GZ Express Chipset (2972), Mobile Intel 956 Express Chipset (2a02)</td>
<td>Chrome (10, 17); Firefox (10)</td>
<td>Windows (7, XP)</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>Intel 82845G (2562), G41 Express Chipset (2e32), 945GM/GU Express Chipset (27a2); Unknown</td>
<td>Chrome (18)</td>
<td>Windows XP</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Intel 82945G Express Chipset (2772); Intel HD Graphics (0102); Mobile Intel 945 Express Chipset (27a2)</td>
<td>Chrome (17, 18); Firefox (8, 10)</td>
<td>Windows (7, Vista, XP)</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>Intel 82945G Express Chipset (2772)</td>
<td>Firefox (8, 10)</td>
<td>Windows (XP)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2.2. Groups of identical WebGL renders. Number in parentheses indicate device IDs, where appropriate. (continued)

<table>
<thead>
<tr>
<th>#</th>
<th>Size</th>
<th>Graphics Cards</th>
<th>Browsers</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>24</td>
<td>Intel G33/G31 Express Chipset (29c2); Intel Graphics Media Accelerator 3150 (a011); Unknown</td>
<td>Chrome (13, 17, 18); Firefox (8, 9, 10, 11)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>Intel G33/G31 Express Chipset (29c2)</td>
<td>Firefox (10)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>24</td>
<td>80</td>
<td>Intel G41 Express Chipset (2e32); Intel Graphics Media Accelerator HD (0046); Intel HD Graphics (0042, 0046, 0df1); Mobile Intel 4 Series (2a42); Mobile Intel 45 Series (0x2a42); Unknown</td>
<td>Chrome (15-19); Firefox (9, 10, 11)</td>
<td>Windows (7, 8, Vista, XP)</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>Intel HD Graphics (0102, 0116)</td>
<td>Firefox (8, 9, 10)</td>
<td>Windows (7, XP)</td>
</tr>
<tr>
<td>26</td>
<td>13</td>
<td>Intel HD Graphics (0102, 0116); Unknown</td>
<td>Chrome (16, 17); Firefox (10, 11)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>27</td>
<td>10</td>
<td>Mobile Intel 965 Express Chipset (2a02, 2a12); Intel GMA X3100 (2a02)</td>
<td>Chrome (14, 17, 18); Firefox (8, 9, 10)</td>
<td>Windows (7, XP)</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>NVIDIA GeForce 7025</td>
<td>Firefox (10)</td>
<td>Linux</td>
</tr>
<tr>
<td>29</td>
<td>19</td>
<td>NVIDIA GeForce 9400M</td>
<td>Firefox (10)</td>
<td>OS X 10.6</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>NVIDIA GeForce GTX 210, 315M, 8400GS, 8500 GT, 8600M GT, 9200M GS, 9500 GT, 9600 GT, 9600M GT, G 105M, G 210M, GTS 250, GTS 360M, GTX 295</td>
<td>Chrome (13, 17); Firefox (10)</td>
<td>Windows (7, Vista)</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>NVIDIA GeForce 315M; Unknown</td>
<td>Firefox (5, 6)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>NVIDIA GeForce 320M</td>
<td>Safari (5)</td>
<td>OS X 10.7.3</td>
</tr>
<tr>
<td>34</td>
<td>6</td>
<td>NVIDIA GeForce 410M, GT 425M, GTX 460, GTX 460 SE</td>
<td>Chrome (17); Firefox (10)</td>
<td>Windows (7, XP)</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>NVIDIA GeForce GTX 550</td>
<td>Chrome (17)</td>
<td>Linux</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>NVIDIA GeForce 6150SE, 6200, 7100, 7150M, 7300 SE/7200GS; Unknown</td>
<td>Chrome (15, 17); Firefox (10)</td>
<td>Windows (7, XP)</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>NVIDIA GeForce 6200</td>
<td>Firefox (4)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>NVIDIA GeForce 6800 Series (00f9)</td>
<td>Firefox (10)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>NVIDIA GeForce 7025</td>
<td>Chrome (18)</td>
<td>Windows XP</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>NVIDIA GeForce 7300SE/7200GS</td>
<td>Firefox (10)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>NVIDIA GeForce 7650 GS</td>
<td>Chrome (17)</td>
<td>Windows 7</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2.2. Groups of identical WebGL renders. Number in parentheses indicate device IDs, where appropriate. (continued)

<table>
<thead>
<tr>
<th>#</th>
<th>Size</th>
<th>Graphics Cards</th>
<th>Browsers</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>4</td>
<td>NVIDIA GeForce 8600, 9800, 320M, 9400</td>
<td>Chrome (17);</td>
<td>Linux;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Firefox (10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OS X (10.6, 10.7.3)</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>4</td>
<td>NVIDIA GeForce 8400 GS, 9500 GT, Quadro FX 1800M; Unknown</td>
<td>Chrome (15, 17, 19);</td>
<td>Windows 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Firefox (13)</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>NVIDIA GeForce GT 220</td>
<td>Firefox (10)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>NVIDIA GeForce GT 330M</td>
<td>Chrome (19)</td>
<td>OS X 10.7.3</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>NVIDIA GeForce GT 440</td>
<td>Opera (9.80)</td>
<td>Windows 7</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>Intel 965GM</td>
<td>Firefox (8)</td>
<td>Linux</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
<td>Intel 965GM</td>
<td>Firefox (10)</td>
<td>Linux</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>Mesa DRI Intel Ironlake Mobile</td>
<td>Firefox (10)</td>
<td>Linux</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>Mesa DRI Intel Sandybridge Mobile</td>
<td>Firefox (10)</td>
<td>Linux</td>
</tr>
<tr>
<td>No</td>
<td>30</td>
<td>AMD Radeon HD 6670, 7450M; ATI Radeon HD 3200, X1200, X1900, XPRESS 200; Intel GMA</td>
<td>Chrome (15, 16, 17);</td>
<td>Linux;</td>
</tr>
<tr>
<td>WebGL</td>
<td></td>
<td>3100; Intel 82865G; Intel G41; Intel HD Graphics; NVIDIA GeForce 315M, 6150SE, 6200, FX</td>
<td>Opera (9.80); Safari (5)</td>
<td>OS X (10.6.8); Windows (7, Vista, XP)</td>
</tr>
</tbody>
</table>
Chapter 3

Are AES x86 Cache Timing Attacks Still Feasible?

We argue that five recent software and hardware developments — the AES-NI instructions, multicore processors with per-core caches, complex modern software, sophisticated prefetchers, and physically tagged caches — combine to make it substantially more difficult to mount data-cache side-channel attacks on AES than previously realized. We propose ways in which some of the challenges posed by these developments might be overcome. We also consider scenarios where side-channel attacks are attractive, and whether our proposed workarounds might be applicable to these scenarios.

3.1 Introduction

Side-channel attacks are a classic topic in computer security. But are they still feasible on modern x86 machines? In a side-channel attack, an attacker recovers secret information from his victim by observing or manipulating a shared resource. The most attractive channels for such attacks are shared hardware resources such as the data cache, and the most devastating attacks recover cryptographic keys. Even as the traditional scenario for side channel attacks — multiuser timesharing on workstation systems — has fallen into decline, other attractive attack scenarios have arisen.
This chapter arises from our unsuccessful attempt at exploiting one such scenario: compromising Chrome browser SSL keys from a Native Client control, using AES cache timing as a channel. In our attempt, we ran up against several recent changes to the x86. Some of these changes have already been mentioned in work on side-channel attacks; others are well known by architects but less so in the security community. Taken together, these changes make it much more difficult to mount side-channel attacks on AES.

Our contribution in this chapter is to describe the new challenges to AES cache attacks and to propose ways in which they might be overcome. We also consider scenarios where side-channel attacks are attractive, and whether our proposed workarounds might be applicable to these scenarios.

The new challenges to AES side-channel attacks are:

- The AES-NI instruction set, which moves AES data structures out of the cache;
- multicore processors with per-core L1 and L2 caches;
- the complexity of modern software and the pressure that it places on caches;
- the increasingly sophisticated and poorly documented prefetcher units on modern processors; and
- the switch from virtually tagged to physically tagged caches.

The first two of these can make AES cache attacks impossible; the last three increase the difficulty of AES cache attacks and make them inapplicable in some settings.

Architectural side-channel attacks are enabled when shared hardware between two mutually distrusting principals is incompletely virtualized by the supervisor. Traditionally, the principals were users on a timesharing system and the supervisor was the OS kernel. We believe that today there are (at least) three scenarios where architectural side-channel attacks are a threat: (1) infrastructure-as-a-service cloud computing [119], with virtual
machines as principals; (2) client-side in the Web browser and its plugins, with Web origins as principals; and (3) smartphones and tablets, with apps as principals. (Mobile devices mostly use ARM chips, not x86.)

Cache side-channel attacks on AES were first demonstrated by Bernstein [15], Tromer, Osvik, and Shamir [139], and Bonneau and Mironov [18]. These attacks were compared and analyzed by Canteaut [27], and were recently improved by Gullasch, Bangerter, and Krenn [53]. Proposed mitigations include cache-oblivious AES algorithms [23], the AES-NI instruction set [66], deterministic computing environments [8], and fuzzy timing channels [145].

In this chapter, we argue that mounting AES (data) cache attacks on modern systems is more difficult than previously realized. This does not, of course, mean that they are impossible. In addition, our results do not rule out attacks on other cryptographic primitives, such as RSA [109], nor attacks that rely on other architectural channels, such as the instruction cache [1, 2] or the branch prediction unit [3], nor timing attacks that do not depend on an architectural channel [24, 16]. In addition, while we have considered the x86, other architectures may still be vulnerable. Viewed another way, our results suggest that other cryptosystems than AES, other microarchitectural features than the data cache, and other architectures than the x86 should be considered in future side-channel research. The recent paper of Zhang et al. [155] partly corroborates this positive view of our claims. Zhang et al. mount a cross-VM side-channel attack on cryptographic keys; but they target public-key crypto rather than AES, and use the instruction cache rather than the data cache.

### 3.2 Complete Mitigation

First, we identify two trends in processor development which have the potential to completely prevent AES cache timing attacks: AES-NI and multicore processors. These
recent phenomenon represent material change in the hardware capabilities underlying our computer systems, and will almost certainly grow for the foreseeable future.

3.2.1 AES-NI

Cache timing side channel attacks depend solely on measuring the processor’s use of memory during encryption. Without these cache-changing accesses, the entire class of attacks is mitigated.

Intel processors that support AES-NI [52] provide hardware implementations of key generation, encryption rounds, and decryption rounds. The cryptographic operations are moved out of RAM and into custom hardware, improving performance and eliminating cache side channels.

Hardware Prevalence

Intel shipped the first AES-NI–supporting processor in Q1 2010 [67]. All Sandy Bridge i7 and all but two Sandy Bridge i5 support AES-NI, while all announced Ivy Bridge i5 and i7 processors support it. Though Intel is still producing a few processors without AES-NI, presumably it will shortly be supported throughout the entire lineup (akin to the MMX or SSE extensions). AMD is introducing AES-NI support as well, with Bulldozer, the first supported microarchitecture, released in October 2011 [49].

Therefore, many consumer-facing systems built in the past two years are inoculated against AES cache timing attacks, and we posit that this number will grow with time.

Even on processors that do not implement AES-NI, alternative AES implementations exist that keep AES data outside of RAM; see, e.g., [101].
Software Support

Of course, all the AES-NI hardware in the world can’t protect against cache timing attacks if applications persist in using vulnerable software AES implementations. Fortunately, SSL and crypto libraries are actively providing support for AES-NI: OpenSSL 1.0.1 [107], Microsoft Cryptography API: Next Generation in Windows 7 [66], and Network Security Services 3.12.2 [100] all utilize AES-NI when the processor provides support. Therefore, up-to-date versions of Google Chrome, Microsoft Internet Explorer, and Mozilla Firefox are all completely immune to AES cache timing attacks on modern hardware.

3.2.2 Multicore Processors

In the pursuit of performance, processor designers are increasingly adding multiple physical cores to each die. For example, currently Intel is almost exclusively shipping multicore chips, with a few single core holdouts in the Celeron and Atom lines. The Atom N270 on which we conducted our experiments contains only a single core, but newer processors in the Intel Atom line (designed to be minimal and power–efficient) now contain multiple cores. Even mobile devices are touting dual (or quad) cores as a major selling point. Multicore is here to stay, and, in order to remain a threat, cache timing attacks must be proven to work under the new hardware regime.

The mere inclusion of multiple cores complicates cache attacks immensely. First, the attack must be aware of processor-specific multicore cache behavior. For example, Intel Sandy Bridge processors have a per-core L1 and L2 cache, but all cores share L3. The attacker must also understand the eviction policy: can data remain in a L2 cache if it is evicted from L3 by another core? Further complicating matters, an attacking thread might be rescheduled onto another physical core at any time, physically separating it from its carefully manicured test data. Therefore, the best approach might take cues
from Gullasch’s attack: create many attacker threads, and trust that attacker threads will probabilistically control every core most of the time. Under this scheme, cross-thread communication is vital: the attacker, when communicating with herself, must take care to avoid the cache lines she’s trying to measure. While a solution to each of these problems is imaginable, none of the existing cache timing attacks are directly applicable to multicore processors, and further work would definitely be needed to indicate that such attacks remain viable in the multicore paradigm.

Recent work by Xu et al. on L2 cache timing side channels in virtualized environments show that the upper bound on side channel bandwidth in EC2 is just over 10 bits per second [151]. A covert channel consists of communication between two cooperating principals; it is not clear that Xu et al.’s analysis can be applied to adversarial side channels, let alone to the very specific side channels required for attacking AES using the data cache.

Furthermore, core “pinning” is becoming an extremely popular technique to manage load and scheduling behavior. Essentially, the operating system assigns a thread to one or more physical cores, and guarantees that it will never execute anywhere else. With this in mind, we observe that a pinned attacker thread (even if pinned to \( n - 1 \) cores) physically cannot perform cache timing attacks against an encryption thread on the forbidden core, especially when that core’s L2 cache vastly exceeds 4 KiB. The AES lookup tables will fit entirely in the encrypting core’s L2, which the attacker is unable to manipulate or examine, rendering the technique powerless.

### 3.3 Attack Outline

The first element needed in a successful cache timing attack is a high resolution timer, able to differentiate the few hundred cycles between a cache hit or miss. For this purpose, our predecessors in cache timing use the x86 instruction `rdtsc`, which provides
cycle-accurate count information. To also use rdtsc, we need to be able to deliver and execute semi-arbitrary x86 instructions on the target machine. Google Chrome provides such an ability in Native Client.

Native Client (NaCl) allows web developers to run native code alongside web applications, sidestepping the performance penalty of interpreters or JITs. It uses sophisticated software-fault isolation techniques to prevent misbehavior. For example, Native Client executables are restricted from making arbitrary system calls, and developer-provided code is restricted from making syscalls at all (special NaCl code, placed in a certain location in the NaCl application’s address space, handles outside communication). Furthermore, writes to memory are region-checked, so applications are prevented from simply writing a stream of instructions and executing them. Control flow is also restricted: jump destinations must always be on an aligned address. Lastly, NaCl disallows the use of certain x86 instructions, such as ret.

NaCl is permissive enough that NaCl code can mount cache timing attacks. NaCl allows use of both the x86 instructions rdtsc and cpuid, a serializing instruction that returns detailed information about processor features. NaCl code can issue an arbitrary sequence of memory reads.

Next, a cache timing attack must cause encryptions to occur, ideally with valuable keys. Assuming the network attacker model, the most attractive use of AES in Google Chrome is to protect TLS traffic. As a network attacker can record encrypted traffic between the victim’s browser and the target domain, possessing the AES key protecting that traffic allows the attacker to decrypt and read every packet in the flow. Therefore, malicious NaCl code should attempt to induce traffic in an TLS session to a domain of interest.

The NaCl standard library provides the URLRequestInfo and URLLoader classes, which, when used in unison, can be used to fetch remote assets for use by NaCl clients.
However, to direct traffic to our third-party victim domain, we must bypass the same origin policy as enforced by the browser. Usefully, when fetching remote resources, NaCl implements Cross-Origin Resource Sharing (CORS)\(^1\). This draft specification lets clients request cross-origin fetches by attaching optional headers to the request, and allows domains to explicitly enable such requests by adding optional headers to the response. Of course, in our attack model, the attacker cannot be certain that the remote domain implements CORS or allows cross-domain fetches from our attacking origin. However, their support (or lack thereof) doesn’t impact the attack: we only need to cause AES encryptions and decryptions. To comply with CORS, Chrome must parse the response’s headers before deciding if the NaCl client is allowed to see the results. So, a NaCl request for a remote resource will always trigger a fetch to be sent over the network. Therefore, by simply requesting any resource in the victim HTTPS origin, a malicious NaCl client can insert traffic into any SSL stream it chooses (even if it will never receive the results of that fetch).

We chose to investigate the feasibility of cache timing attacks on the Intel Atom N270 processor. Released in 2008 as one of the first Atom chips, the N270 is a relatively simple 32-bit single core hyperthreaded processor with a 512 KiB, 8-way associative L2 cache. Importantly, it lacks AES-NI hardware (as discussed in Section 3.2.1).

And so, \textit{prima facie}, the possibility for a cache timing attack using Native Client against AES as used by Google Chrome seems quite good. However, in trying to implement such an attack, we have discovered three major roadblocks standing between us and a successful exploit on the N270: modern software engineering, prefetching, and cache indexing. Since we feel that these comprise a material setback against the viability of cache timing attacks in the wild, we present them in the next few sections.

\(^1\)Online: http://w3.org/TR/2012/WD-cors-20120403/
3.4 Modern Software Engineering

Unfortunately for a cache attacker, real-world programs do not perform thousands of AES operations and then immediately cease execution. At the very least, encrypted data will be written to disk or the network, incurring operating system complexity and overhead. Even local functions called before or after the AES operation require the processor to fetch instruction memory touched by the program counter, causing even more cache evictions and noise.

Therefore, to provide a lower bound on the cache noise that our attacker must disregard, we examine the internal architecture and code size of Chromium, the open source version of Google Chrome. First, we acquired and built Chromium r105554 in debug mode, and created a minimal NaCl application which causes a cross-origin HTTPS fetch. Next, we attached GDB to various parts of Chromium as the fetch was triggered, and wrote a script which, through judicious use of the step and disassemble GDB commands, revealed the size and location of each function called by Chromium during each period of execution. This provides a slight overestimate of actual utilized code size (since portions of each function are undoubtedly skipped by, e.g., an if statement), but this measurement completely ignores any data structures, including the stack, heap, global variables, and C++ virtual method tables. It also does not count any operating system overhead, such as process scheduling, memory mapping, handling interrupts, or facilitating inter-process communication. Therefore, we believe that this metric underestimates the total cache-impacting memory fetches during Chromium’s execution.

3.4.1 Chromium Architecture

Chromium, to facilitate security guarantees, uses sandboxing mechanisms to separate responsibilities among several processes. For our purposes, we must inspect
three distinct processes: the browser, the renderer, and the NaCl runtime. The browser is the main thread, responsible for managing all network and disk access, while the renderer is a origin-specific sandbox which handles parsing HTML/CSS and executing JavaScript in order to render a particular page. The NaCl runtime exists in a third process, where the NaCl application is loaded and run, and this runtime communicates solely with its parent renderer. When our attacking NaCl code initiates an asynchronous HTTPS fetch, the request first travels via RPC into the renderer, where it is checked for viability. Next, the renderer makes another RPC to the controlling browser process, requesting the fetch. At that point, the browser opens a socket and sleeps. When the OS reports that socket is writable, a browser thread wakes up and finally performs the AES encryption.

### 3.4.2 Measurements

We attached GDB to the renderer process, at `NaClSrpcReceiveAndDispatch()`. While there are several more layers of calls above this one handling other tasks of the RPC, this is the function which initiates the URL loading process. Allowing GDB to run and disassemble all functions until this function returns, we find that the renderer thread calls at least 2288 distinct functions, totaling 282 KiB and spread across 484 memory pages.

Next, we attached GDB to the browser process, at `HttpNetworkTransaction::OnStreamReady()`. We then perform our step-and-disassembly analysis down the entire call stack, recording which functions are called. Note, again, that this underestimates the code size as well, since there are a few more general functions above `OnStreamReady()` in the call stack. Nevertheless, Chromium executes at least 894 distinct functions, encompassing 165 KiB on 254 pages.

Lastly, we examine the overhead of receiving a packet of TLSed data, with the idea that an attacker could initiate a very long download over HTTPS and asyn-
 chronously attack AES while the download completes. To do this, we set a breakpoint on SSLClientSocketNSS::BufferRecvComplete and executed until it returned. During this operation, 158 functions were executed for 29 KiB on 82 pages.

### 3.5 Prefetching

Designed to reduce memory latency and increase performance, hardware prefetchers speculate about future memory accesses and issue fetch requests for likely data. Unfortunately for our purposes, these fetches cause replacements and evictions on cache lines that would be otherwise untouched during AES operations.

First described by Baer and Chen in 1991 [9], stride prefetchers attempt to notice memory accesses at constant offsets (such as traversing an array) and issue requests for the next items in the series. This behavior wreaks havoc on naïve cache probing strategies — simply accessing 8 memory locations which map to the same cache index might trigger the ninth to be fetched, evicting one of the original 8. Tromer et al. describe a clever workaround: a linked list is created using the locations of each intended probing address, connecting every address in a random order. As the list is traversed, the stride prefetcher does not detect a regular pattern of memory accesses, and does not trigger fetches.

As processors evolve and gain extra transistors, prefetchers become more complicated. Since prefetchers greatly impact processor performance and interesting new prefetching strategy can be implemented without breaking compatibility, processor manufacturers do not release precise descriptions of their prefetchers’ behavior. Security analyses usually assume access to the full specification of the system under attack, but reverse-engineering the prefetcher circuit from a 32 nm process chip would be prohibitively expensive.

Our experiments with the Atom N270 indicate that its hardware prefetcher does, in fact, influence the viability of cache timing attacks. In Figures 3.1 and 3.2, some of
the prefetching behavior of the processor is revealed. To generate these timings, we first flush the cache, then access memory in a few dis-contiguous cache lines. Then, using \texttt{rdtsc}, we measure the access time for an individual cache line. The cache is then reset and the process repeated for the next location. Each location was measured 16 times.

Examining Figure 3.1, we see that, as expected, accesses to cache lines 0, 4, and 8 always hit in the cache. Surprisingly, lines 1, 5, and 6 always hit as well, indicating that the prefetcher is always active. We also see occasional hits at locations 9 through 12, and 15, indicating further prefetching. In Figure 3.2, the prefetcher is even more active. Lines 0, 1, and 3 through 8 always hit, even though we prime only four lines. Cache hits are also occasionally seen at higher locations, indicating unpredictable prefetching behavior.

To avoid this complex prefetching behavior, the attacker can, in theory, access relatively distant cache lines. However, since fetching cache line $A$ always brings in line $A + 1$ (see lines 0 and 1 in figures), our cache timing resolution is limited to 128 bytes, as accessing one 64 B cache line brings in its 64 B neighbor. When probing 1 KiB AES tables on the N270 (even if prefetching behavior is reverse-engineered and understood), cache measuring techniques offer at most 3 bits of table offset, as opposed to the 4 bits supposed by both Tromer and Gullasch.

Interestingly, though, hardware prefetchers can be disabled, and are often shipped disabled on server-class hardware. Note, however, that disabling the prefetcher requires either modifications to the BIOS or an instruction run at privilege level 0, and if an attacker can effect either of these things, they have full control over the hardware; a cache timing attack is superfluous. However, in a server environment, AES cache timing attacks may be able to sidestep the effects of a hardware prefetcher.
Figure 3.1. Access times per cache line after cache flush on Atom N270 (min, average, max across 16 runs). Timing done after accesses at lines 0, 4, then 8.

Figure 3.2. Access times per cache line after cache flush on Atom N270 (min, average, max across 16 runs). Timing done after accesses at lines 0, 3, 4, then 8.

Figure 3.3. Address bits on Atom N270
3.6 Cache Indexing

The basic operation of any cache timing attack is the cache line probe. To do so, current cache timing attacks load their own data into the cache and measure, through timing repeated accesses, when said data is evicted from the cache. As one of the first steps towards implementing the attack, then, we tested how this might work on our Atom N270.

The L2 cache in the Atom N270 is 512 KiB, with a line size of 64 B and an 8-way set associativity. With these characteristics, the cache contains 1024 distinct sets, each able to contain 8 lines. Since each line is 64 bytes long, the memory address will be broken down into 6 tag bits, 10 index bits, and 16 tag bits (shown in Figure 3.3).

Therefore, addresses which differ only in their top 16 bits should map to a single cache set. Also, once we access 9 or more of these addresses, we should begin to see evictions and memory access times should spike.

To test this hypothesis, we created a program which begins by loading 1 MiB of useless data into the cache. This provides a fair baseline from which to start testing, since any memory location that we will test should now be evicted. Next, we access \( n \) memory locations at a stride of 16 KiB, taking care to use a non-linear order designed to minimize the effects of a stride prefetcher. Once this is done, we access all \( n \) locations again in the same order, measuring each memory access time, in cycles. We then ran this program on Linux 3.0.0 and FreeBSD 9.0. Figure 3.4 plots the average access time, across four runs, for each location in this second round against the total number of accesses.

As expected, when the number of locations is small, our entire data set fits in the cache, and accesses are fast. However, once we access 9 unique locations, there is a drastic performance difference between Linux and FreeBSD. On Linux, we continue to see access times in line with L2 cache hits; FreeBSD shows the expected behavior of memory fetches.
The explanation for this disparate behavior lies not only in the physical hardware of the N270, but in the virtual memory subsystems of each operating system as well. FreeBSD, unlike most modern operating systems, implements “cache coloring” [39], which attempts to allocate virtual memory so that virtual pages correspond to physical pages. That is, cache coloring allows processes to predict which pages will conflict in the cache, and optimize their memory accesses accordingly. Linux 3.0 and OS X do not implement cache coloring. In order to understand how cache coloring impacts our attack, however, we must discuss how the processor manages its caches.

The N270, along with almost every modern x86 CPU, has “physically-indexed” caches, or, more specifically, caches which slot memory locations into sets using their physical addresses, rather than their virtual ones. Since there is no mechanism for a process to discover the physical addresses of its memory (which, indeed, might change at any point), physically-indexed caches offer no exploitable guarantees about which addresses will map to the same cache set. This uncertainty increases the difficulty and complexity of mounting any sort of cache timing side channel attack.

**Figure 3.4.** Average access time for virtual addresses which should collide in the cache. Error bars indicate the maximum and minimum time seen.
3.6.1 Attack Complexity

From the physical capabilities of the N270, we can predict exactly how much more difficult an attack becomes on a physically-tagged processor. The N270, along with most other x86 processors, provides support for 4 KiB memory pages. When a virtual address is mapped to its physical counterpart, the lower \( \log_2(4 \text{KiB}) = 12 \) bits remain unchanged. From Figure 3.3, the N270 uses address bits 15 through 6 to index into a cache set. Therefore, for any given virtual address on the N270, 6 of 10 index bits are already known, leaving 4 unknown index bits.

To predict how these unknown bits complicate an attack against Google Chrome’s SSL AES implementation, we examine the position and size of the AES lookup tables. On Ubuntu Linux 11.10, Chromium links against and uses the OS’s NSS library. Since Ubuntu Linux implements ASLR, we cannot predict the page’s location in virtual memory (or, indeed, its physical tag), but the page offset will be constant! The AES lookup tables, in Chromium’s address space, always begin at a page offset of 0xA0. Since the tables total 4 KiB, the area of interest to our AES cache attack spans two physical pages. As such, the attacker must now discover the physical positions of both these pages, among the \( 2^{4+4} = 2^8 \) possible configurations. Since there are no direct methods of probing these tables, the attacker is reduced to either triggering AES operations and searching the cache for likely positions, or by simply guessing at the tables’ position in physical space.

We note, as well, that when our attacking thread runs on physically-indexed caches, there is no reliable way to determine which addresses in its own virtual address space map to identical cache lines. However, by the pigeonhole principle, once the attacking thread possesses \( (1024 \times 8 + 1) = 8193 \) pages of memory (a manageable 32 MiB), at least one cache set is guaranteed to contain at least 9 pages, and the attacker can conceivably discover, by eviction, which of their pages overlap in the cache.
3.7 Conclusions

In conclusion, we present an re-analysis of the computational complexity of a cache timing attack against AES with a key size of 128 bits, modeled after that of Tromer. Their analysis is detailed and precise but, as a full summary will not fit in this chapter, we make do with an abbreviated version. First, let $\delta$ be the cache line size divided by the size of a table entry (4 bytes). Due to the well-meant meddling of the prefetcher, the effective distinguishable cache line size on the Atom N270 is not the physically-defined $64\text{B}$, but rather $128\text{B}$, so $\delta = 32$. For comparison, in Tromer’s initial analysis, $\delta' = 16$.

Next, Tromer’s analysis assumes access to an ideal predicate $Q_k(p, l, y)$, which is defined to be 1 iff, during the AES encryption, the block $y$ in lookup table $T_l$ is accessed at least once, for key $k$ and plaintext $p$. With this predicate, an attack against the simplest first round of AES reveals the top $\log_2(256/\delta) = 3$ bits of each key byte.

Next, Tromer applies a attack against the second round of AES. From analysis of the AES algorithm, they find that four table lookups in the second round can be directly computed from four key bytes. Applying $Q_k(p, l, y)$ with $l = 2$, they compute the table accesses across every possible key. In their analysis, since $\delta' = 16$, this gives an analysis complexity of $\delta'^4 = 2^{16}$ — easily manageable. In our case, however, $\delta^4 = 2^{20}$. This process must then be repeated four times, since each iteration reveals the lower bits of four bytes, and our key is 16 bytes long. To sample $Q_k(p, l, y)$ enough for this analysis to function requires about $\log \delta^{-4}/\log(1 - \delta/256 \cdot (1 - \delta/256)^{38})$ samples, which for our $\delta$ is 17,720.

Therefore, our second round attack has a full computational complexity of $4 \cdot 2^{20} \cdot 17,720 \approx 2^{36}$ to extract the full AES key. Already, this is looking dire.

Compounding the issue further, $Q_k(p, l, y)$ is an ideal predicate — we must make do with an approximation. Furthermore, we must take into account the cost of finding the
AES tables in memory, given the fact that modern processors are exclusively physically-tagged. Without some clever trick, we posit that the best an attacker can do is guess, for both pages, where in the cache they lie. Since each page has 4 unknown bits on the Atom N270, we must repeat our analysis $2^{4+4} = 2^8$ times, once with each guess. This brings our total expected work to $2^{44}$.

Finally, as if it weren’t hard enough, the attack must be able to pick out the AES page table accesses from the avalanche of memory fetches caused by simply executing the target program. Whenever a NaCl client requests a remote resource, approximately 447 KiB of code is accessed. Within a 512 KiB cache, this represents a serious disturbance to any prepared cache timing attacker. Attempting to extract the effects of a few thousand accesses to a region of 4 KiB in an unknown location appears to be a herculean effort.

Therefore, we posit that any data-cache timing attack against x86 processors that does not somehow subvert the prefetcher, physical indexing, and massive memory requirements of modern programs is doomed to fail, to say nothing of the difficulties imposed by multicore processors and hardware AES implementations.

**Acknowledgments**

Chapter 3, in part, is a reprint of the material as it appears in ACM Cloud Computing Security Workshop (CCSW) 2012. Mowery, Keaton; Keelveedhi, Sriram; Shacham, Hovav, 2012. The dissertation author was a primary investigator and the primary author of this paper.
Chapter 4

Welcome to the Entropics: Boot-Time Entropy in Embedded Devices

We present three techniques for extracting entropy during boot on embedded devices. Our first technique times the execution of code blocks early in the Linux kernel boot process. It is simple to implement and has a negligible runtime overhead, but, on many of the devices we test, gathers hundreds of bits of entropy. Our second and third techniques, which run in the bootloader, use hardware features — DRAM decay behavior and PLL locking latency, respectively — and are therefore less portable and less generally applicable, but their behavior is easier to explain based on physically unpredictable processes.

We implement and measure the effectiveness of our techniques on ARM-, MIPS-, and AVR32-based systems-on-a-chip from a variety of vendors.

4.1 Introduction

Random numbers unpredictable by an adversary are crucial to many computing tasks. But computers are designed to be deterministic, which makes it difficult to generate random numbers. Substantial effort has gone into developing and deploying subsystems that gather and condition entropy, and that use it to generate random numbers on demand.
In this chapter, we take an extreme position: Randomness is a fundamental system service; a system cannot be said to have successfully booted unless it is ready to provide high-entropy randomness to applications.

Our main contributions are three techniques for gathering entropy early in the boot process — before interrupts are enabled, before a second kernel thread is spawned. Our techniques are suitable for use even on embedded systems, where entropy-gathering is more challenging than on desktop PCs. We implement our proposed techniques and assess their effectiveness on systems-on-a-chip (SoCs) that integrate ARM, MIPS, and even AVR32 CPU cores.

Motivation

Our work is inspired by the recent paper of Heninger, Durumeric, Wustrow, and Halderman [59], which uncovered serious flaws in the design and implementation of the Linux kernel’s randomness subsystem. This subsystem exposes a blocking interface (/dev/random) and a nonblocking interface (/dev/urandom); in practice, nearly all software uses the nonblocking interface. Heninger et al. observe (1) that entropy gathered by the system is not made available to the nonblocking interface until Linux estimates that 192 bits of entropy have been gathered, and (2) that Linux is unnecessarily conservative in estimating the entropy in events, and in particular that on embedded systems no observed events are credited with entropy. These two facts combine to create a “boot-time entropy hole,” during which the output of /dev/urandom is predictable.

The Linux maintainers overhauled the randomness subsystem in response to Heninger et al.’s paper. The timing of every IRQ is now an entropy source, not just IRQs for hard disks, keyboards, and mice. Entropy is first applied to the nonblocking pool, in the hope of supplying randomness to clients soon after boot. (Clients waiting on the blocking interface can block a bit longer.)
The new design leaves in place the race condition between entropy accumulation and the reading of supposedly random bytes from the nonblocking pool. It would be better, we argue, to gather entropy so early in the boot process that all requests for randomness can be satisfied.

In this chapter, we present entropy-gathering techniques that realize this vision. We show how to gather entropy in the bootloader or early in the kernel boot process on embedded systems running a variety of popular processors. Our techniques require neither the multicore x86 processor of desktop PCs nor the sophisticated sensors available to smartphones. They do not require network connectivity. They can be used in place of, or side by side with, Linux’s current entropy-gathering infrastructure.

Our three techniques provide different tradeoffs along three metrics: (1) How many random bits can be obtained, and how quickly? (2) How much system-specific knowledge is required to implement the technique? (3) To what extent can the entropy obtained be explained by well-studied physical processes that are believed to be unpredictable? None of our proposed techniques is ideal along all three metrics.

**Our first technique: Instruction timing early in kernel boot**

We instrument the kernel’s startup to record how long each block of code takes to execute. This approach has previously been used to gather entropy in userland code; we show that it is also applicable for a single kernel thread of execution, with interrupts disabled, on an embedded system. On many of the devices we tested (see Section 4.2), this technique gathers a surprisingly large amount of entropy — over 200 bits on the Raspberry Pi — at negligible runtime overhead; on other devices, less entropy is available.

We have not been able to account conclusively for the large amount of entropy this technique gathers on some devices or for the smaller amount it gathers on other devices. In Section 4.3, we pinpoint architectural features that are partly responsible.
Our second and third techniques: DRAM decay and PLL locking

In our second class of techniques, we take advantage of architectural features that vary between SoCs, rendering them less portable and less widely applicable, but promising more entropy. In addition, we are able to pinpoint more precisely the sources of the entropy we measure.

In Section 4.4, we show that it is possible for bootloader code, running from on-chip SRAM, to turn off DRAM refresh. With refresh disabled, the contents of DRAM decay unpredictably; we exploit this fact to obtain an entropy source. In Section 4.5, we show that our ability to repeatedly reconfigure a peripheral clock on the BeagleBoard xM translates into another high-rate entropy source.

4.1.1 Related Work

As noted above, the motivation for our chapter is Heninger et al.’s recent study of the Linux randomness subsystem [59].

Random number generation is hard, and flaws in randomness subsystems have been identified with dismaying regularity. In 1996, Goldberg and Wagner analyzed the random number generator in the Netscape browser [51]. A decade later, Luciano Bello found that the OpenSSL package shipped with Debian and Ubuntu had a broken random number generator [135]. The bug’s effects were quantified by Yilek et al. [154]. Cryptographers have designed “hedged” cryptosystems whose security degrades as little as possible in the absence of good randomness [13]. Otherwise secure random number generators can break in novel settings: Ristenpart and Yilek observed that virtual machine resets could lead to randomness reuse and proposed solutions [120, 153].

Researchers have expended considerable effort considering how best to design randomness subsystems. Gutmann described design principles for random number gen-
erators [54]; Kelsey, Schneier, Wagner, and Hall proposed a formal security model for random number generators and described attacks on deployed systems [84]. Kelsey, Schneier, and Ferguson then proposed Yarrow, a concrete design for a family of random number generators [83]. More recently, NIST has made recommendations for producing random numbers from an entropy pool [11]. Researchers have also studied the effectiveness of the randomness subsystems deployed with Linux [55, 90] and Windows [41]. Gutterman, Pinkas, and Reinman, in their study of Linux randomness system [55] specifically pointed out the vulnerability of Linux-based routers like those running OpenWRT software.

Entropy can be obtained from many sources: from dedicated hardware, from using analog feedback circuits such as phase-locked loops (PLLs) [46] or digital feedback circuits (as included in Intel’s latest processors [57, 22]); from timing other hardware devices, such as hard disks [36, 72]; from timing user input; or, in sensor-rich devices such as smartphones, from sensor noise in microphones [42, Section 5.3.1], cameras [20], and accelerometers [146].

Instruction timings have long been used as a source of entropy. In Section 4.2.1 we describe Bernstein’s `dnscache-conf` program from 2000. The method was explored in detail in the HAVEGE system of Seznec and Sendrier [125]. In both cases, the entropy is assumed to derive from the unpredictable arrival times of interrupts and the behavior of the system scheduler. By contrast, our first technique (described in Section 4.2) obtains entropy even with interrupts disabled and a single thread of execution.

Pyo, Pae, and Lee, in a short note, observe that DRAM refresh timings are unpredictable, which means that DRAM access timings can be used as an entropy source [113].

Theoretical grounding for the unpredictability of instruction timing was given by McGuire, Okech and Zhou [96] and Mytkowicz, Diwan, and Bradley [104]. These papers consider x86 chips; the processors we study are considerably simpler.
Decay patterns in RAM, used in our second technique (described in Section 4.4), have also been considered before. Holcomb et al. use SRAM decay as an entropy source on RFID devices [61]. Halderman et al. studied DRAM decay patterns in detail [56].

4.2 Early Kernel Entropy

Our first method for gathering entropy is an application of a simple idea: After each unit of work in a code module, record the current time using a high-resolution clock. Specifically, we instrument `start_kernel`, the first C function run in the Linux kernel on boot, and use the cycle counter as our clock.

Our approach is attractive. It runs as early as possible in the kernel boot process: All but one use of randomness in the Linux kernel occurs after `start_kernel` has completed. It imposes almost no performance penalty, requiring, in our prototype implementation, 3 KiB of kernel memory and executing a few hundred assembly instructions. It is simple, self-contained, and easily ported to new architectures and SoCs.

The question is, Does it work? Previous applications of the same idea ran in user mode on general-purpose x86 machines. They could take advantage of the complexity of the x86, the unpredictable arrival timing of interrupts, interleaved execution of other tasks, and the overhead of system call servicing when accessing a high-resolution clock. By contrast, our code runs on an embedded device with interrupts disabled and a single thread of execution. Nevertheless, we are able to extract a surprising amount of entropy — in some cases, hundreds of bits.

In this section, we discuss our implementation and evaluate its effectiveness on ARM SoCs from six vendors, a MIPS SoC, and an AVR32 SoC. In Section 4.3, we discuss architectural mechanisms that are partly responsible for the entropy we observe.
4.2.1 Genesis

In 2000, Daniel J. Bernstein released dnscache 1.00, a caching DNS recursive resolver that is now part of the djbdns package. DNS resolvers generally operate over UDP, which means that an interested attacker can spoof the answer to a query by simply forging a packet. To combat this, each DNS query carries along with it a pre–selected port number and query ID, which the response must have to be considered valid. Therefore, dnscache, when acting as a client of other DNS servers, must be able to choose these port numbers and query IDs well [80, 64].

One of dnscache-conf’s duties is to provide entropy that will later be used by dnscache. To gather this entropy, the dnscache-conf utility simply instruments its own startup procedure with multiple calls to gettimeofday(), and mixes each result into the entropy pool. Due to the cost of each syscall, unpredictable hardware interrupts, OS process scheduling, clock skew, and a host of other factors, this method provides dnscache-conf with high-quality entropy for the cost of a few extra syscalls. An excerpt from dnscache-conf.c:

```c
makedir("log");
seed_addtime();
perm(02755);
seed_addtime();
makedir("log/main");
seed_addtime();
owner(pw->pw_uid,pw->pw_gid);
seed_addtime();
perm(02755);
seed_addtime();
```
A method which works in userland on an x86 machine might not apply to kernel-level code on much simpler embedded devices. Indeed, we were initially skeptical: In the absence of interrupts, multiple threads, syscall overhead, and on simpler processors than the x86, would there still be enough variation to make such a scheme viable?

4.2.2 Methodology

Kernel Instrumentation

To collect information about the kernel boot process, we modified a Linux kernel for each system we examined. Our kernel instrumentation consists of a basic macro that can be inserted anywhere in kernel boot to record the current cycle count with low overhead. The macro recorded the current cycle count to an incrementing index in a statically allocated array. We incremented the index at compile time, and thus the only operations performed by the measurement at run time are reading the cycle counter and a single memory store.

We inserted the macro between every function call in `start_kernel`, the first C function called during kernel boot. The majority of the code executed during this sequence is straight-line, with a varying number of instructions executed during each function call. We chose this sampling method because it offered the simplest patch to the kernel at the earliest point in the boot process. Our instrumentation then printed the measured times to the kernel log. An init script copied out the relevant data from the log, truncated the log, and immediately restarted the system using `reboot`. Temperature data was not collected. In this manner, we gathered data on thousands of restarts per day per machine with minimal interaction. Machines were switched off and the data pulled after 24–48 hours of continuous rebooting and data collection.

To estimate the performance overhead, we implemented a “production-ready” version, which skips printing to the kernel log in lieu of mixing the results directly into
the kernel’s randomness pools. We then used the cycle counter to measure the execution
time of `start_kernel`, both with and without our instrumentation. On the Raspberry
Pi (detailed in Section 4.2.3), our technique adds approximately 0.00019 seconds to the
kernel boot process.

**Devices**

As described in the previous section, we instrumented a variety of Linux kernels
and ran them on a broad variety of embedded platforms, ranging from high-powered
ARM computers to low-end special-purpose MIPS and AVR devices.

**ARM**

ARM, Inc. licenses its processor architecture to many companies that integrate
ARM cores into their designs. Two systems-on-a-chip that integrate the same ARM core
might nevertheless have very different performance characteristics. To check the general
applicability of our approach to ARM-based embedded systems, we instrumented and
collected data from systems-on-a-chip from many of the most prominent ARM licensees:
Broadcom, Marvell, NVIDIA, Texas Instruments, Qualcomm, and Samsung. These
vendors represent six of the top seven suppliers of smartphone processors by revenue.

Specifically, the first system we tested was the Raspberry Pi, which contains a
Broadcom BCM2835 SoC featuring a 1176JZF-S core, which is an ARM11 core imple-
menting the ARMv6 architecture. We also instrumented the BeagleBoard xM, which uses
a Texas Instruments DM3730 containing a ARMv7 Cortex-A8; the Trim-Slice featuring
an NVIDIA Tegra 2, a ARMv7 Cortex-A9; the Intrinsyc DragonBoard, with a Qualcomm
SnapDragon SoC containing a Qualcomm Krait ARMv7; the FriendlyARM Mini6410
with a Samsung S3C6410, another version of the ARM1176JZF-S ARM11 ARMv6 core;
and the Cubox, with a Marvell ARMADA 510 SoC containing a Sheeva ARMv7 core.
MIPS

Previous work on embedded device entropy identified routers as important targets, as they are conveniently located to inspect and modify network traffic and, as reported by Heninger et al. [59], routinely ship with extremely poor entropy, as evidenced by their SSL certificates.

With this in mind, we instrumented the early Linux boot process on the Linksys WRT54GL router, containing a Broadcom 5352EKPBG 200MHz MIPS “router-on-a-chip.” Revered for their extensibility, the WRT54GL represents a basic wireless router as found in the homes of millions.

AVR32

Finally, we instrumented a kernel for the Atmel NGW100 mkII, which contains a AT32AP7000-U AVR32 core. The AVR32, designed by Atmel, represents one of the smallest and lowest-power CPUs capable of running Linux. Even on the AVR32, our techniques uncover substantial randomness. The existence of instruction entropy on this platform indicates that execution randomness is not solely due to processor optimizations and complexity.

4.2.3 Results and Analysis

In this section, we will discuss the results of each device’s kernel instrumentation, and the expected quality of the entropy extracted.

As the existence of true randomness is an open philosophical question (and therefore beyond the scope of this chapter), we will treat entropy as “unpredictability”: given the knowledge that a remote attacker can possibly have, how difficult would it be to guess the device–generated random bits?
Statistical Tests

We are unable to conclusively pinpoint and characterize every source of entropy in these systems. Therefore, this analysis will deal only with empirical measurements, as sampled from each board over many boots. We will rely mainly on two estimations: distribution entropy and min-entropy.

Distribution entropy represents, for a given empirical sample, the Shannon entropy of the underlying distribution. For example, a set of samples consisting of 50 A’s and 50 B’s would have a single bit of distribution entropy, while a set of samples consisting of 1024 unique values has a distribution entropy of 10 bits. Distribution entropy can be calculated, for a set $S$ of $n$ distinct observed values $V\_i$, each being seen $C\_i$ times, with $C = |S| = \sum_{i=0}^{n} (C\_i)$, as:

$$D(S) = -\sum_{i=1}^{n} \frac{C\_i}{C} \cdot \lg \left( \frac{C\_i}{C} \right)$$

Note that distribution entropy will almost always underestimate the entropy of the underlying distribution. That is, the distribution entropy calculated from a empirical sampling $S$ will always be less than or equal to $\lg (|S|)$, regardless of the actual entropy of the underlying distribution.

Our other empirical estimator, min-entropy, measures the prevalence of the most common element in a distribution. In other words, if an adversary is allowed a single guess at the value, min-entropy measures how often she will be correct. For a set $S$ of $n$ distinct observed values $V\_i$ with counts $C\_i$, the min-entropy is:

$$M(S) = -\lg \left( \frac{\max_{i}(C\_i)}{C} \right)$$

With these two metrics, we can characterize the distributions sampled from each device and predict their real-world entropy content.
Entropy Extraction

Furthermore, each boot sequence generates a vector of test times, one per test. In our analysis, we will examine both the sampled distributions of individual test times, as well as the sampled distribution of test vectors. The test vector, once generated, can be fed into an entropy extractor to produce an evenly–distributed random seed, which can then used to seed kernel pseudo-random number generators.

The values in the test vector are partly correlated: if nothing else, later tests have cycle counts larger than earlier tests. Extracting the entropy from such a source is a challenging theoretical problem [106], but under the random oracle heuristic simply applying a cryptographic hash to the test vector is sufficient. NIST has published explicit recommendations for implementing what they call “reseeding” in randomness generators [11].

Raspberry Pi

The Raspberry Pi is a popular single-board ARM computer, built around the Broadcom BCM2835 System–on–a–chip (SoC), which contains an ARM 1176JZF-S ARMv6 core clocked at 700 MHz. We modified the Linux 3.2.27 kernel provided for the Raspberry Pi¹ to perform our data collection. This involved enabling and configuring the hardware cycle counter and the two hardware performance counters present in the ARM 1176JZF-S, as well as surrounding each function call in start_kernel with instrumentation to record the current counter values, and a final function to dump our results to the kernel log. We were able to surround every function in start_kernel, for a total of 78 individual tests.

Next, we booted the instrumented kernel on four identical Raspberry Pis, and recorded the counters for each boot.

¹Online: https://github.com/raspberrypi/linux
In short, almost every test shows a surprising amount of variation in the number of cycles it takes to execute. Figures 4.1, 4.2, 4.3, and 4.4 show a histogram of test times, in cycles, for tests 4, 5, 36, and 41 as seen across 301,647 boots across all four Raspberry Pi devices. The lighter regions are the contribution of device #0, which, by itself, contributes 130,961 boots. These four graphs are representative of the four classes of histogram that we see on the Raspberry Pi: a “two-normal” distribution like Test 4, a “quantized” distribution like Test 5, a “bimodal plus noise” distribution like Test 36, and a “normal” distribution like Test 41.

For comparison, Test 4 corresponds to the initialization function `cgroup_init_early()`, which is responsible for setting up process groups for resource management, and mostly involves setting memory locations to initial values. Test 5 is `local_irq_disable()`, which disables interrupts. It consists solely of the ARM instruction "cpsid i", and the variation in this test is likely due to hardware initialization state. Test 36 is `prio_tree_init()`, and is simply a small loop which initializes an array. The relatively quantized period of this function is likely due to stalls in memory.

**Figure 4.1.** Histogram of cycle counts for Test 4 on 4 Raspberry Pis. Lighter region is data from device #0 only.
Figure 4.2. Histogram of cycle counts for Test 5 on 4 Raspberry Pis. Lighter region is data from device #0 only.

Figure 4.3. Histogram of cycle counts for Test 36 on 4 Raspberry Pis. Lighter region is data from device #0 only.
fetches and stores. Also, note that IRQs remain disabled until Test 45, and so interrupts cannot be blamed for any variation in these test times.

Overall, these distributions are far wider than we initially expected. Test 41, in particular, has a minimum value of 69,098 cycles and a maximum of 76,625, almost 10.9% more. In this region of 7,527 cycles, the data set contains 5,667 distinct test values.

Taken individually, the results of each test give an empirical distribution over the cycles elapsed during execution. If we treat a test as a random variable, we can extract that entropy and use it to seed a random number generator.

To estimate the entropy contribution of each test, we apply the distribution entropy calculation to our observed data. The results of this calculation are in Table 4.1.

However, further investigation is needed before we proclaim success. While each test has between 0.45 and 12.99 bits of distribution entropy, we cannot naively sum these numbers and proclaim that to be our total entropy produced. In order for that approach to be valid, each test must be statistically independent — the time taken for test $T$ must not depend on the results for tests $(0, \ldots, T-1)$. If, in the worst case, $T$ was a known

\[ \text{Figure 4.4. Histogram of cycle counts for Test 41 on 4 Raspberry Pis. Lighter region is data from device #0 only.} \]
Table 4.1. Per-Test Distribution Entropy, in bits

<table>
<thead>
<tr>
<th>Test</th>
<th>RPi</th>
<th>BB</th>
<th>Trim-Slice</th>
<th>Dragon-Board</th>
<th>Mini-6410</th>
<th>Cubox</th>
<th>WRT</th>
<th>NGW</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>4.07</td>
<td>4.10</td>
<td>5.70</td>
<td>9.48</td>
<td>-</td>
<td>0.12</td>
<td>8.14</td>
<td>4.33</td>
</tr>
<tr>
<td>#1</td>
<td>1.91</td>
<td>7.31</td>
<td>4.31</td>
<td>4.30</td>
<td>0.55</td>
<td>5.02</td>
<td>6.82</td>
<td>0.78</td>
</tr>
<tr>
<td>#2</td>
<td>0.85</td>
<td>2.32</td>
<td>4.77</td>
<td>2.33</td>
<td>0.-</td>
<td>0.-</td>
<td>7.25</td>
<td>1.80</td>
</tr>
<tr>
<td>#3</td>
<td>8.51</td>
<td>2.58</td>
<td>9.97</td>
<td>10.34</td>
<td>0.29</td>
<td>0.42</td>
<td>4.66</td>
<td>0.78</td>
</tr>
<tr>
<td>#4</td>
<td>9.32</td>
<td>8.76</td>
<td>10.88</td>
<td>9.51</td>
<td>5.41</td>
<td>0.-</td>
<td>4.66</td>
<td>0.78</td>
</tr>
<tr>
<td>#5</td>
<td>2.58</td>
<td>2.78</td>
<td>2.95</td>
<td>2.74</td>
<td>0.17</td>
<td>0.-</td>
<td>4.15</td>
<td>0.77</td>
</tr>
<tr>
<td>#6</td>
<td>6.58</td>
<td>2.33</td>
<td>8.86</td>
<td>5.04</td>
<td>0.51</td>
<td>1.21</td>
<td>8.04</td>
<td>2.36</td>
</tr>
<tr>
<td>#7</td>
<td>5.25</td>
<td>5.45</td>
<td>7.66</td>
<td>5.44</td>
<td>3.28</td>
<td>2.12</td>
<td>8.80</td>
<td>1.05</td>
</tr>
<tr>
<td>#8</td>
<td>2.24</td>
<td>3.85</td>
<td>9.45</td>
<td>8.77</td>
<td>0.34</td>
<td>2.45</td>
<td>2.21</td>
<td>1.80</td>
</tr>
<tr>
<td>#9</td>
<td>8.79</td>
<td>5.09</td>
<td>10.98</td>
<td>8.97</td>
<td>6.03</td>
<td>4.55</td>
<td>3.57</td>
<td>2.19</td>
</tr>
<tr>
<td>#10</td>
<td>13.42</td>
<td>9.32</td>
<td>11.28</td>
<td>14.21</td>
<td>7.74</td>
<td>7.92</td>
<td>6.60</td>
<td>2.19</td>
</tr>
<tr>
<td>#11</td>
<td>2.73</td>
<td>11.42</td>
<td>8.54</td>
<td>4.01</td>
<td>1.78</td>
<td>0.78</td>
<td>7.87</td>
<td>2.19</td>
</tr>
<tr>
<td>#12</td>
<td>2.73</td>
<td>6.25</td>
<td>7.97</td>
<td>2.47</td>
<td>6.34</td>
<td>0.-</td>
<td>7.99</td>
<td>2.19</td>
</tr>
<tr>
<td>#13</td>
<td>8.48</td>
<td>5.27</td>
<td>9.77</td>
<td>8.63</td>
<td>0.03</td>
<td>1.68</td>
<td>8.70</td>
<td>1.00</td>
</tr>
<tr>
<td>#14</td>
<td>2.28</td>
<td>9.82</td>
<td>8.67</td>
<td>6.06</td>
<td>7.47</td>
<td>0.-</td>
<td>0.78</td>
<td>1.58</td>
</tr>
<tr>
<td>#15</td>
<td>12.19</td>
<td>7.54</td>
<td>10.79</td>
<td>10.85</td>
<td>0.79</td>
<td>2.41</td>
<td>0.41</td>
<td>1.00</td>
</tr>
<tr>
<td>#16</td>
<td>2.00</td>
<td>11.12</td>
<td>7.95</td>
<td>4.96</td>
<td>6.93</td>
<td>0.-</td>
<td>1.59</td>
<td>0.78</td>
</tr>
<tr>
<td>#17</td>
<td>9.83</td>
<td>6.40</td>
<td>9.95</td>
<td>10.20</td>
<td>0.70</td>
<td>1.52</td>
<td>6.22</td>
<td>0.78</td>
</tr>
<tr>
<td>#18</td>
<td>3.88</td>
<td>10.04</td>
<td>8.38</td>
<td>8.44</td>
<td>6.95</td>
<td>0.-</td>
<td>9.92</td>
<td>1.00</td>
</tr>
<tr>
<td>#19</td>
<td>8.48</td>
<td>8.27</td>
<td>9.40</td>
<td>7.67</td>
<td>7.30</td>
<td>0.01</td>
<td>9.09</td>
<td>0.78</td>
</tr>
<tr>
<td>#20</td>
<td>11.43</td>
<td>9.21</td>
<td>10.43</td>
<td>6.65</td>
<td>7.32</td>
<td>2.53</td>
<td>8.66</td>
<td>1.00</td>
</tr>
<tr>
<td>#21</td>
<td>3.24</td>
<td>6.15</td>
<td>3.80</td>
<td>11.12</td>
<td>6.74</td>
<td>0.-</td>
<td>7.28</td>
<td>1.00</td>
</tr>
<tr>
<td>#22</td>
<td>6.30</td>
<td>10.59</td>
<td>8.59</td>
<td>6.37</td>
<td>1.27</td>
<td>1.05</td>
<td>8.66</td>
<td>0.78</td>
</tr>
<tr>
<td>#23</td>
<td>10.85</td>
<td>6.63</td>
<td>10.18</td>
<td>9.72</td>
<td>7.60</td>
<td>1.50</td>
<td>9.34</td>
<td>1.00</td>
</tr>
<tr>
<td>#24</td>
<td>13.52</td>
<td>5.89</td>
<td>10.51</td>
<td>9.68</td>
<td>6.82</td>
<td>1.14</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>#25</td>
<td>9.85</td>
<td>10.55</td>
<td>10.28</td>
<td>10.83</td>
<td>4.94</td>
<td>1.84</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#26</td>
<td>2.29</td>
<td>10.17</td>
<td>6.26</td>
<td>4.05</td>
<td>7.73</td>
<td>0.-</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>#27</td>
<td>14.15</td>
<td>8.47</td>
<td>11.29</td>
<td>14.43</td>
<td>7.73</td>
<td>2.90</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>#28</td>
<td>8.14</td>
<td>4.85</td>
<td>10.41</td>
<td>10.24</td>
<td>4.07</td>
<td>1.77</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>#29</td>
<td>3.80</td>
<td>11.77</td>
<td>8.68</td>
<td>4.97</td>
<td>7.36</td>
<td>0.-</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#30</td>
<td>9.60</td>
<td>10.20</td>
<td>10.01</td>
<td>2.21</td>
<td>7.22</td>
<td>1.99</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#31</td>
<td>10.29</td>
<td>3.74</td>
<td>10.03</td>
<td>10.15</td>
<td>7.56</td>
<td>2.07</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#32</td>
<td>5.14</td>
<td>3.20</td>
<td>10.05</td>
<td>9.73</td>
<td>7.57</td>
<td>0.-</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#33</td>
<td>9.28</td>
<td>9.89</td>
<td>9.54</td>
<td>10.27</td>
<td>4.79</td>
<td>2.18</td>
<td>-</td>
<td>5.90</td>
</tr>
<tr>
<td>#34</td>
<td>11.83</td>
<td>9.52</td>
<td>11.02</td>
<td>8.24</td>
<td>6.33</td>
<td>2.78</td>
<td>-</td>
<td>0.02</td>
</tr>
<tr>
<td>#35</td>
<td>11.07</td>
<td>10.49</td>
<td>10.76</td>
<td>12.70</td>
<td>4.94</td>
<td>3.57</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#36</td>
<td>7.67</td>
<td>9.81</td>
<td>7.94</td>
<td>11.00</td>
<td>4.67</td>
<td>1.12</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#37</td>
<td>9.21</td>
<td>10.41</td>
<td>10.29</td>
<td>4.96</td>
<td>7.07</td>
<td>2.12</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#38</td>
<td>7.53</td>
<td>11.31</td>
<td>9.25</td>
<td>8.45</td>
<td>7.53</td>
<td>1.26</td>
<td>-</td>
<td>0.78</td>
</tr>
<tr>
<td>#39</td>
<td>7.38</td>
<td>9.24</td>
<td>8.68</td>
<td>4.43</td>
<td>0.64</td>
<td>0.10</td>
<td>-</td>
<td>0.14</td>
</tr>
<tr>
<td>#40</td>
<td>8.15</td>
<td>8.07</td>
<td>11.17</td>
<td>6.28</td>
<td>7.53</td>
<td>2.96</td>
<td>-</td>
<td>0.12</td>
</tr>
<tr>
<td>#41</td>
<td>11.67</td>
<td>7.36</td>
<td>10.81</td>
<td>8.78</td>
<td>4.50</td>
<td>5.49</td>
<td>-</td>
<td>5.07</td>
</tr>
<tr>
<td>#42</td>
<td>7.02</td>
<td>8.88</td>
<td>8.41</td>
<td>13.75</td>
<td>1.35</td>
<td>1.04</td>
<td>-</td>
<td>4.27</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Test</th>
<th>RPi</th>
<th>BB</th>
<th>Trim-Slice</th>
<th>Dragon-Board</th>
<th>Mini-6410</th>
<th>Cubox</th>
<th>WRT</th>
<th>NGW</th>
</tr>
</thead>
<tbody>
<tr>
<td>#43</td>
<td>2.21</td>
<td>11.55</td>
<td>9.24</td>
<td>3.55</td>
<td>9.74</td>
<td>0.74</td>
<td>-</td>
<td>1.25</td>
</tr>
<tr>
<td>#44</td>
<td>3.88</td>
<td>7.99</td>
<td>6.43</td>
<td>6.65</td>
<td>4.20</td>
<td>0.93</td>
<td>-</td>
<td>1.90</td>
</tr>
<tr>
<td>#45</td>
<td>4.27</td>
<td>9.08</td>
<td>6.70</td>
<td>1.06</td>
<td>0.82</td>
<td>0.74</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#46</td>
<td>10.97</td>
<td>4.33</td>
<td>9.05</td>
<td>1.62</td>
<td>2.82</td>
<td>5.57</td>
<td>-</td>
<td>6.00</td>
</tr>
<tr>
<td>#47</td>
<td>11.36</td>
<td>3.22</td>
<td>10.67</td>
<td>9.65</td>
<td>0.65</td>
<td>3.85</td>
<td>-</td>
<td>5.35</td>
</tr>
<tr>
<td>#48</td>
<td>2.47</td>
<td>4.00</td>
<td>0.06</td>
<td>2.79</td>
<td>1.05</td>
<td>0.74</td>
<td>-</td>
<td>3.55</td>
</tr>
<tr>
<td>#49</td>
<td>2.46</td>
<td>6.24</td>
<td>3.48</td>
<td>2.89</td>
<td>0.92</td>
<td>0.74</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#50</td>
<td>2.64</td>
<td>11.24</td>
<td>8.03</td>
<td>10.36</td>
<td>0.70</td>
<td>0.08</td>
<td>-</td>
<td>2.28</td>
</tr>
<tr>
<td>#51</td>
<td>2.64</td>
<td>7.09</td>
<td>5.91</td>
<td>1.96</td>
<td>8.14</td>
<td>0.74</td>
<td>-</td>
<td>2.02</td>
</tr>
<tr>
<td>#52</td>
<td>2.58</td>
<td>5.47</td>
<td>7.08</td>
<td>1.49</td>
<td>2.34</td>
<td>0.74</td>
<td>-</td>
<td>3.01</td>
</tr>
<tr>
<td>#53</td>
<td>1.79</td>
<td>5.25</td>
<td>6.97</td>
<td>5.16</td>
<td>9.93</td>
<td>0.74</td>
<td>-</td>
<td>2.80</td>
</tr>
<tr>
<td>#54</td>
<td>3.19</td>
<td>6.23</td>
<td>7.38</td>
<td>3.03</td>
<td>9.96</td>
<td>2.75</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#55</td>
<td>10.68</td>
<td>4.88</td>
<td>10.12</td>
<td>3.62</td>
<td>12.88</td>
<td>0.08</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#56</td>
<td>4.65</td>
<td>4.60</td>
<td>3.31</td>
<td>1.91</td>
<td>11.21</td>
<td>0.02</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#57</td>
<td>5.16</td>
<td>4.26</td>
<td>7.99</td>
<td>1.78</td>
<td>10.48</td>
<td>5.37</td>
<td>-</td>
<td>3.83</td>
</tr>
<tr>
<td>#58</td>
<td>6.65</td>
<td>10.05</td>
<td>5.11</td>
<td>8.85</td>
<td>3.09</td>
<td>5.06</td>
<td>-</td>
<td>3.25</td>
</tr>
<tr>
<td>#59</td>
<td>12.56</td>
<td>5.37</td>
<td>10.91</td>
<td>6.19</td>
<td>8.04</td>
<td>3.51</td>
<td>-</td>
<td>2.52</td>
</tr>
<tr>
<td>#60</td>
<td>11.23</td>
<td>5.44</td>
<td>10.33</td>
<td>2.50</td>
<td>9.09</td>
<td>1.68</td>
<td>-</td>
<td>3.25</td>
</tr>
<tr>
<td>#61</td>
<td>9.62</td>
<td>5.81</td>
<td>10.09</td>
<td>2.01</td>
<td>10.23</td>
<td>3.15</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#62</td>
<td>4.60</td>
<td>11.76</td>
<td>8.93</td>
<td>11.67</td>
<td>8.88</td>
<td>2.83</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#63</td>
<td>8.05</td>
<td>10.61</td>
<td>9.10</td>
<td>9.53</td>
<td>0.88</td>
<td>4.03</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#64</td>
<td>8.92</td>
<td>10.44</td>
<td>9.97</td>
<td>9.50</td>
<td>0.54</td>
<td>2.03</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#65</td>
<td>10.42</td>
<td>5.80</td>
<td>9.85</td>
<td>3.60</td>
<td>1.04</td>
<td>0.06</td>
<td>-</td>
<td>1.28</td>
</tr>
<tr>
<td>#66</td>
<td>8.79</td>
<td>8.17</td>
<td>9.22</td>
<td>8.20</td>
<td>11.73</td>
<td>0.74</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#67</td>
<td>8.55</td>
<td>8.97</td>
<td>9.41</td>
<td>9.80</td>
<td>9.63</td>
<td>0.74</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#68</td>
<td>1.73</td>
<td>9.86</td>
<td>10.71</td>
<td>9.94</td>
<td>6.87</td>
<td>4.15</td>
<td>-</td>
<td>0.74</td>
</tr>
<tr>
<td>#69</td>
<td>11.03</td>
<td>9.40</td>
<td>10.45</td>
<td>9.07</td>
<td>11.69</td>
<td>3.70</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#70</td>
<td>12.99</td>
<td>8.20</td>
<td>10.99</td>
<td>7.87</td>
<td>11.20</td>
<td>2.06</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#71</td>
<td>10.42</td>
<td>10.76</td>
<td>10.02</td>
<td>3.20</td>
<td>1.99</td>
<td>4.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#72</td>
<td>8.50</td>
<td>10.52</td>
<td>9.52</td>
<td>2.46</td>
<td>0.99</td>
<td>0.74</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#73</td>
<td>10.92</td>
<td>11.15</td>
<td>10.62</td>
<td>12.03</td>
<td>0.77</td>
<td>0.74</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#74</td>
<td>11.79</td>
<td>9.75</td>
<td>10.70</td>
<td>9.79</td>
<td>12.28</td>
<td>0.68</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#75</td>
<td>2.45</td>
<td>9.57</td>
<td>8.82</td>
<td>8.94</td>
<td>1.63</td>
<td>0.74</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#76</td>
<td>9.48</td>
<td>11.23</td>
<td>9.93</td>
<td>10.73</td>
<td>1.22</td>
<td>4.71</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#77</td>
<td>9.45</td>
<td>-</td>
<td>9.70</td>
<td>11.07</td>
<td>1.26</td>
<td>0.74</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Sum</td>
<td>564.98</td>
<td>594.66</td>
<td>683.40</td>
<td>557.84</td>
<td>394.78</td>
<td>129.15</td>
<td>151.41</td>
<td>90.21</td>
</tr>
</tbody>
</table>
function of \((0, \ldots, T - 1)\), then it would not contribute any entropy whatsoever to the overall total, even if it had plenty of distribution entropy: Its distribution entropy would already be counted by the preceding tests. (Note that we can always safely mix the results of \(T\) into the entropy pool. In the worst case, doing so adds no benefit.)

We applied a straightforward correlation test to the data we gathered from the Raspberry Pi and our other systems. More sophisticated tests are possible, for example using NIST’s test suite [122]. Specifically, we computed the correlation coefficients between each pair of tests. We can then select a threshold of acceptable risk, and exclude from our entropy estimate any tests which are correlated with another test beyond that limit. Figure 4.5 shows the full entropy estimate function for the Raspberry Pi for all possible thresholds. This function is surprisingly linear, suggesting that the Raspberry Pi tests, while correlated, do not cluster together when taken as a whole. With no self-evident threshold to pick, we arbitrarily exclude from our entropy estimate any tests having correlation of 0.4 or more with another test.
Figure 4.6. Cycle counts for Tests 4 and 7 on the Raspberry Pi. Correlation coef. = −0.79. Line is the best-fit linear model.

Applying this test to our Raspberry Pi data, we find some intriguing results. Figure 4.6 shows a scatterplot of Test 4 and Test 7. The former is the kernel function `cgroup_init_early`, which is responsible for initializing resource–managing process groups, and mainly consists of initializing variables throughout kernel memory. The latter, on the other hand, is `boot_cpu_init`, which is in charge of marking the CPU as “online” and “active”, allowing other cores to communicate. (Note that the Raspberry Pi has only a single core, but still executes this step.) We have so far been unable to determine a causal relationship between these two tests that might account for the extremely odd relationship in Figure 4.6.

While we do not believe that the correlations between tests are particularly helpful to an attacker (since a remote or local but post-boot attacker will not have access to the preceding $T - 1$ test values), in the interests of caution, we modify our entropy estimate as follows: for each successive variable, add its distribution entropy to the total if and only if, when correlated with each preceding variable in turn, never has a correlation
coefficient with magnitude $\geq 0.4$. If the variable is thus correlated with a preceding variable, we ignore its sampled distribution entropy entirely.

When computed across the entire Raspberry Pi data set, this conservative estimate places the summed distribution entropy of pairwise uncorrelated variables at 231.9 bits — far beyond the reach of exhaustive-search attacks.

Finally, to ensure that this analysis is not completely off, we compute the distribution entropy over the entire data set of 79-element vectors. For the 301,647 Raspberry Pi boot measurements in our data set, every single one is unique, giving a distribution entropy of 18.2 bits. Since distribution entropy cannot extrapolate beyond the size of the empirical data set, this is an empirical lower bound on the entropy available by simply instrumenting the boot procedure of Linux on the Raspberry Pi, and, given our calculations above, we believe that there is more than sufficient entropy available during the Raspberry Pi’s boot process to securely seed the Linux randomness generator.

**BeagleBoard xM**

The BeagleBoard xM is powered by a Texas Instruments DM3730 SoC, containing a 1 GHz Cortex-A8 ARMv7 superscalar CPU core. We acquired and modified a patched Linux 3.2.28-x14\(^2\) to include 77 tests in `start_kernel`.

We have less Linux boot data for the BeagleBoard than our other systems, as we re-purposed the BeagleBoard for other experiments, detailed in Section 4.4. Nevertheless, we collected data on 3,580 boots.

Per-test distribution entropies for the BeagleBoard are in Table 4.1. Naively summing, these 77 tests give 594.66 bits of entropy between them. Our correlation coefficient threshold test reduces this slightly, to 430.06 bits. As for empirical distribution entropy, all 3,580 boot sequences are unique, giving a distribution entropy floor of 11.81 bits.

\(^2\)Online: https://github.com/RobertCNelson/stable-kernel
Trim-Slice

The Trim-Slice is another ARM single-board computer, designed for use as a desktop PC. It contains a 1 GHz NVIDIA Tegra 2 dual-core Cortex-A9 ARMv7 CPU, and a variety of storage options. To stay consistent with our other devices, we chose to boot the Trim-Slice from its MicroSD slot. We modified a Linux 3.1.10-l4t.r15.02 kernel\(^3\) to include our instrumentation, and set the machine to rebooting. Our particular model had an issue of failing to reboot every so often, limiting our data collection for this device.

Nevertheless, we instrumented 2,522 reboots of the Trim-Slice, collecting cycle counts for 78 tests, similar to the Raspberry Pi kernel. Per-test distribution entropy can be found in Table 4.1, giving a total sum of 683.40 bits (which, again, may not be an accurate total estimate). Interestingly, even though the Trim-Slice data set contains 100 times fewer boots than the Raspberry Pi data, the per-test distribution entropies are roughly similar across the board. Since distribution entropy chronically underestimates the entropy of the underlying distribution, this implies that the Trim-Slice’s Tegra 2 has a much wider test variation than the ARM 1176JZF-S, which is eminently plausible given that the Tegra 2 is a dual-core platform and based on a Cortex-A9, a larger and more complex core than in the Raspberry Pi.

The Trim-Slice tests also appear to show much less correlation than the Raspberry Pi. When we apply our method of summing only the distribution entropy of variables which are not pairwise correlated with any previous test (cor. coef. ≤ 0.4), the Trim-Slice tests still show a shocking 641.48 bits of entropy. Even if this overstates the actual amount by a factor of 3, there is easily enough entropy extractable on boot to seed any pseudorandom generator.

Finally, as one might expect given the data thus far, each of the 2,522 78-element test vectors sampled on a given Trim-Slice boot is unique, giving a total distribution

\(^3\)Online: https://gitorious.org/trimslice-kernel
entropy of 11.30 bits. Again, this represents an empirical lower bound, and is one which we believe is extremely low.

**Intrinsyc DragonBoard**

The Intrinsyc DragonBoard is a fully-featured mobile device development board based around the Qualcomm Snapdragon S4 Plus APQ8060A SoC, which includes a Qualcomm Krait ARMv7 dual-core CPU. Designed as a development board for Android mobile devices, it includes hardware such as a touch screen, wi-fi radio, and a camera module.

As a mobile device development platform, the DragonBoard runs Android 4.0.4 and is backed by an Intrinsyc-modified Linux 3.0.21 kernel. As a result, our patch set was easy to apply. As usual, we inserted 78 tests into `start_kernel`. Instead of a Linux init script for collecting the data, we used the Android adb tool to connect to the device via USB, dump the kernel logs and reboot the device. In this way, we collected data on 27,421 boots.

In general, we see excellent entropy generation when booting Linux on the Krait. The per-test distribution entropies can be found in Figure 4.1, with a per-test sum of 557.84 bits. As with our preceding three ARM SoCs, each boot sequence is unique, giving an empirical distribution entropy of 14.74 bits. The tests are also highly uncorrelated: applying our correlation coefficient threshold test lowers the entropy estimate only slightly to 523.55 bits.

Resource-rich embedded devices, such as phones, have a plethora of available sources of entropy — for example, simply turning on one of their radios. This test, though, shows that our entropy generation technique can protect these devices as well.
**FriendlyARM Mini6410**

The FriendlyARM Mini6410 is yet another single-board ARM device. This particular unit is powered by a Samsung S3C6410A SoC, and contains a ARM 1176JZF-S ARM11 core clocked at 533 MHz. As before, we modified the Linux 2.6.38 manufacturer-provided kernel to instrument `start_kernel`, and inserted 77 tests.

Next, we let the FriendlyARM reboot 46,313 times. Interestingly, the data from the FriendlyARM differs significantly from our other ARM results.

First, the per-test distribution entropies for the FriendlyARM can be found in Table 4.1. (The FriendlyARM tests are offset by one to align identical kernel initialization functions between devices as much as possible.) At first glance, the per-test distribution entropies seem reasonable, given that they are bounded above by $\lg(46,313) = 15.4$ bits, naively summing to 394 bits of entropy.

The oddness arrives when we examine the distribution entropy across boot vectors, and not just individual test measurements. Unlike most other ARM SoC we tested, the FriendlyARM occasionally produces identical boot vectors on multiple independent boots. The two most common vectors each appear 15 times in the dataset, giving a min-entropy of 11.59 bits. In other words, a sufficiently prepared adversary, given a single guess, can correctly predict the FriendlyARM’s boot vector with probability $2^{-11.59}$, or about 1 in 3,000. Given 233 guesses, this probability rises to $2^{-4.7}$, or about 1 in every 25. However, this probabilistic defect does not render our instrumentation worthless. Fifty-five percent of vectors in the data set are unique, meaning that this method can fully protect the Linux randomness generator on the FriendlyARM over half the time, for a negligible cost during kernel initialization. Even if the machine does boot with a more common state, mixing in these measurements can never reduce the amount of entropy available to the pool, and thus will never be harmful to the system as a whole.
One might hypothesize that there is some “common” vector, and the other popular vectors are simply approximations thereof. However, the two most popular vectors differ in 59 of 77 positions. Also, strangely, the Mini6410 contains the same ARM core as the Raspberry Pi, which exhibits none of these symptoms. We can find no convincing explanation for the observed difference between these two systems.

**Cubox**

The Cubox is a commercially available desktop platform, powered by the Marvell ARMADA 510 SoC with an 800 MHz Sheeva ARMv7 superscalar CPU core. We modified a Linux 3.6.9 kernel for the Cubox, as before, to include 78 tests. We then rebooted the Cubox 27,421 times.

Per-test distribution entropy for the Cubox is presented in Table 4.1. Interestingly, it is our only ARM SoC which has constant-time tests, i.e., tests whose distribution entropy is zero. It also presents less test entropy overall, with a sum of only 129.15 bits of individual test distribution entropy.

Like the FriendlyARM, the Cubox creates non-unique boots; the most common of these occurs 80 times (0.29%). Only 7,857 boots are unique in our data set. The total empirical distribution entropy of the data set is 12.53 bits, which indicates that our technique, while not solving the entropy-at-boot problem on the Cubox, will still help protect the kernel’s entropy generation.

**Linksys WRT54GL**

While ARM-based embedded devices and SoCs are becoming more and more popular, any investigation into entropy on embedded devices would be remiss without examining how well proposed techniques apply to the large installed base of devices. Home routers, which were recently shown to have insufficient entropy for certificate

---

4Online: https://github.com/rabeeh/linux.git
generation [59], represent an enormous number of existing devices, and, perhaps more importantly, embedded devices where strong entropy sources are extremely important (e.g., key generation). We examined the Linksys WRT54GL as our representative test platform.

The WRT54GL is a popular consumer 802.11B/G wireless router, and consists of a Broadcom BCM5352 “router-on-a-chip”, which contains a 200 MHz MIPS32 core; 16 MiB of RAM; and 4 MiB of flash. Importantly for our purposes, Linksys provides a custom Linux 2.4.20 kernel which can be modified and run on the device.

The stripped-down WRT54GL kernel has fewer function calls in `start_kernel` than the more modern kernels on our ARM boards, but this is to be expected given the simplicity of the device: the kernel needn’t contain any extraneous code. We are able, then, to insert 24 tests in the kernel initialization.

We then ran our modified kernel on two separate WRT54GLs, one unmodified at 200 MHz and one overclocked to 250 MHz. The unmodified WRT we rebooted 81,057 times, while we rebooted the overclocked device 54,465 times. The per-test distribution entropies for the unmodified device are in Table 4.1. Perhaps surprisingly for this device, these per-test entropies are quite high, up to 10 bits in some cases.

However, the correlations between tests on the WRT54GL are far more intertwined than they are on our preceding ARM devices. Two plots of these correlations can be seen in Figures 4.7 and 4.8.

Unfortunately, the overall entropy performance of the WRT54GL betrays its promising per-test entropies. Across the 81,057 boots of our unmodified router, we only see 11.86 bits of distribution entropy, and the most common boot sequence appears 1,247 times (10.4%). Indeed, the top 188 vectors make up 37.1% of the dataset (30,062 boots). If this were the only source of entropy for a PRNG seed, a motivated attacker could easily brute-force these few vectors and succeed almost 40% of the time. Even worse, there are only 11,960 boot sequences we saw only once. If the attacker simply checks the 4,209
Figure 4.7. Cycle counts for Tests 18 and 19 on a WRT54GL. Each point is one boot. Line is best-fit linear model.

Figure 4.8. Cycle counts for Test 0 and 11 on an WRT54GL. Each point is one boot. Line is best-fit linear model.
vectors that she saw more than once during her precomputation, she will succeed against 78.6% of boots.

This unfortunate distribution shows that boot–time entropy is insufficient to protect a PRNG on a standard MIPS home router. However, it does add somewhat more than 11.86 bits, which is our observed distribution entropy across the 24-element test result vectors. Since the process relies solely on an already–extant hardware counter and is virtually free, adding it to the Linux kernel boot is still worthwhile.

**Overclocking**

To see if we could tease further entropy from the WRT54GL, we tried overclocking it from 200 MHz to 250 MHz, on the theory that we could change the ratios between clocks in different parts of the SoC and RAM. On this modified device, we performed 54,465 reboots. Overclocking does materially change each test’s distribution: the Kolmogorov-Smirnov test for distribution equality reports $D > 0.1, P < 2.2 \cdot 10^{-16}$ for 19 of 24 tests, indicating that the two device’s empirical test values are drawn from different underlying distributions. However, the overclocked processor shows the same type of grouping as the unmodified system, giving only 10.4 bits of distribution entropy over the 24-element boot vectors, with the most common appearing 879 times (1.6%).

**Atmel NGW100 mkII**

Finally, we turn to the Atmel NGW100 mkII, a development board for the AT32AP7000-U AVR32 microprocessor. AVR32 processors are designed to be small, low-cost, and low-power: in some sense, it’s one of the smallest microprocessors capable of running Linux. Designed to prototype network gateway devices, the NGW100 mk II ships with multiple Ethernet connectors, internal flash storage, and an SD card slot. To maintain consistency, we booted the NGW100 mkII off an SD card.
We modified and built a patched Linux 2.6.35.4 kernel using the Atmel AVR32 Buildroot system, adding 69 tests to \texttt{start\_kernel}. Then, via an RS-232 console, we rebooted the board 38,157 times. The per-test distribution entropy can be found, as usual, in Table 4.1.

As befits our hypothesis that simpler processors produce more constant results, 28 of the 69 tests have absolutely no variation at all. Most of these functions are simply empty, as the AVR32 is simple enough to not need their services (e.g., \texttt{setup\_nr\_cpu\_ids} is a no-op, as there are no multi-core AVR32 systems), but others do various memory initialization tasks. The constant execution time of these functions speaks to the minimal nature of the system as a whole.

Perhaps not surprisingly, this simplicity takes a toll on the entropy generated during the boot process. Indeed, in our data set, we see only 418 unique 69-element boot vectors; the least frequent of which appears 43 times (0.1%), while the most frequent appears 314 times (0.8%). This suggests rather strongly that we have collected every possible variation of test times the device will generate under standard operating conditions. The empirical distribution entropy of our data set is 8.58 bits; this is likely all the entropy that can be extracted from timing the NGW100 mkII boot.

### 4.3 Architectural Causes of Timing Variation

In this section, we describe two physical mechanisms that partly explain the non-determinism we measured during the execution of early kernel code: communication latency (variation that can arise while sending data between two clock domains) and memory latency (variation that arises due to interactions with DRAM refresh). We give evidence that these mechanisms are involved. We stress that these two mechanisms only partly explain the behavior we observed. Other mechanisms we do not understand are likely also involved; we hope that future work can shed more light on the situation.
4.3.1 Clock domain crossing

Modern embedded processors contain multiple clock domains, and due to misalignment between the domains, the amount of time it takes to send messages between two clock domains can vary.

Processor designs use multiple clock domains to allow different portions of the chip to run at different frequency. For instance, on the BeagleBoard xM, the ARM Cortex-A8 runs at 1 GHz, the peripheral interconnect runs at 200 MHz, and the Mobile DDR memory runs at 166 MHz [134].

At each boundary between two domains, chip designers must use specialized mechanisms to ensure reliable communication. The most common solution to this problem is an asynchronous FIFO (or queue) that enqueues data according to one clock and dequeues it according to a second clock.

To see how an asynchronous FIFO can give rise to latency variation, consider the case when the FIFO is empty and the output domain tries to dequeue data at the same moment the input domain inserts data. If the input domain’s clock arrives first, the dequeue will succeed. If the output domain’s clock arrives first, the dequeue fails and will occur one clock period later. If they arrive at precisely the same time, metastability can result, which will also result in delay. Because of random (and mostly independent) variation in when the two clock signals arrive at the asynchronous FIFO (i.e., clock jitter), any of these orderings is possible and communication latency will vary.

Interactions between different clocks and metastability are well-known sources of very high-quality randomness [86, 130, 34], so it is tempting to try to exploit the domain crossing that already exist in a design to generate random bits.

In order to observe to interactions between clocks on our device, we instrumented code to measure the latency of communication between different clock domains. On
Figure 4.9. Clock domains similar to the domains found on the DM3730. In order for the processor to modify a register in the Ethernet controller, it must cross the clock domains of the first and second level interconnects.

The BeagleBoard xM, there are two on-chip-buses that connect peripherals, similar to the diagram on Figure 4.9. The processor, peripherals and interconnects are clocked by several different PLLs. For the processor to communicate with peripherals on the SoC, the processor must cross these clock domains. Our approach was to measure the variation in latency in communication devices with an increasing number of clock domain crossings. Specifically, we measured the number of cycles it took to perform to complete a set of instructions which did not cross the interconnect (two NOP instructions), to cross the first level interconnect (reading the revision number register of the memory controller) and to cross the second level interconnect (reading the revision number register of the system clock controller).

Our results are shown in Figure 4.10. Variability in frequency increases with the number of clock domains crossed. At two clock domain crossings, the distribution is bimodal. While there may be some serial correlation between repeated runs, this indicates that a read from the second level interconnect can provide up to around 2 bits of entropy. Reads from this register are also fast: at an average of 270 cycles, millions of these reads can be performed each second.
Figure 4.10. Execution latency for two NOP instructions (NOP), a read from the general purpose memory controller (GPMC), and a read from the clock manager (CM). Cycle delta is the difference from the minimum cycles observed.

4.3.2 DRAM Access Latency

A second source of variation in performance is interactions between main memory (i.e., DRAM) accesses, DRAM refresh, and the memory controller. Because DRAM bits decay over time, the system must periodically read and re-write each DRAM storage location. Depending on the system, the processor’s memory controller issues refresh commands or, alternately, the processor can place the chips in an auto-refresh mode so they handle refresh autonomously.

Regardless of who manages it, the refresh process cycles through the DRAM row-by-row, and an incoming memory access (e.g, to fetch part of the kernel for execution) may have to wait for the refresh to complete.

To measure the effect of refresh on execution timing, we used hardware performance counters to measure the number of cycles it took to execute a series of 64 NOP instructions on a ARM Cortex-A9 [148] with the instruction cache disabled 100,000 times. We then used a software register to turn refresh off and performed the test again.
Figure 4.11. Power and clocks on the startup of a typical embedded system. At 1, the voltage is ramped up until it is stable, which can take a variable amount of time. At 2, the system oscillator is turned on and takes a variable amount of time to stabilize. At 3, the PLLs that source the high frequency clocks for the processor (PLL_OUT) and memory (MEM_PLL) are turned on and take a variable amount of time to stabilize. At 4, the time that the memory clock and processor clocks cross is variable but fully determined by the time that both PLLs stabilize. At 5, a small amount of jitter in the memory clock causes the position the clocks cross to change.

The results of our test are plotted in Figure 4.12. The variation in execution latency was much greater with refresh turned on: with refresh on, execution fell into 6 bins, with $\approx 80\%$ distributed at the mode and $\approx 20\%$ distributed in the other 5 bins. With refresh off, over 99\% of executions fell into the mode with less than 1\% distributed in two other bins.

While refresh itself may appear to induce random distributions in our experiment, the state machines in the memory controller and the DRAM chips that manage refresh are deterministic, as is the execution of code on the CPU that generates requests. If the rest of the system were deterministic as well, we expect that DRAM accesses would have deterministic latencies.
However, other sources of randomness can affect the relationship between the processor and the DRAM refresh state machines. For instance, the PLL for the DRAM controller may “lock” more quickly than the processor’s PLL at system (see ④ in Figure 4.11) boot or the DRAM controller’s power supply may take longer to stabilize at start up (see ① in Figure 4.11). In this case, the future interactions between the processor and refresh state machine will be affected, and the latency for DRAM accesses will vary slightly. In addition to variation in the system’s initial conditions, randomness from clock domain crossing can further perturb the interaction between the processor and memory.

### 4.4 DRAM Decay

Ultimately, the most useful source of randomness we found in these system is the decay of data stored in DRAM over time. DRAM decay occurs when the charge that stores a binary value leaks off the capacitor in a DRAM storage cell. This phenomenon is well-studied, and the refresh process (described in 4.3.2) is designed to mitigate it.
Figure 4.13. Decay of DRAM after 7 (Blue), 14 (Green), 28 (Yellow) and 56 (Red) seconds.

Figure 4.14. Probability of decay, per bit, for non-refresh times of 7s, 14s, and 28s. Ordered by Pr(decay) at 28s.
4.4.1 Disabling Refresh

In order to detect decay in a live system, we must prevent the system from refreshing DRAM. The ability to disable refresh on the memory controller is not an exotic feature: Nearly every memory controller we looked at supported disabling refresh, and every embedded SoC we looked at, from the Broadcom BCM5352 found in the WRT54GL to the DM3730 on the BeagleBoard xM had software tunable parameters for controlling refresh [148, 134]. Typically, control over refresh is used to implement sleep modes. When a processor enters a sleep mode, it disables refresh on the memory controller and sends a command to the DRAM chips to enter “self-refresh” mode, forcing the DRAM chips refresh themselves as needed. By turning off refresh on the memory controller and not sending the “self-refresh” command, we were able to observe decay in our test systems.

4.4.2 Decay

The decay rate of DRAM bits varies widely (a fact exploited by “cold boot” techniques [56]) as a result of manufacturing variation, temperature, the data stored in the cell, and other factors. In our experiments, some bits will decay quickly (e.g., on the order of hundreds of µs) while others will retain their data for seconds or hours. We find that the rate at which bits decay varies with even small variations in temperature (see Section 4.4.4).

4.4.3 Experimental Setup

Our approach to harvesting randomness from DRAM is as follows. Very early in the boot process (i.e., in the boot loader) we write test data to a portion of DRAM, and then disable the refresh mechanism in both the processor’s memory controller and the DRAM chips. The processor then waits several seconds, reads the data back, and
Figure 4.15. Probability of decay after one minute.

XORs it with pattern written initially. Any flipped bits will appear at this stage. After this process, the bootloader can re-enable refresh, reinitialize DRAM, and continue loading as normal.

Next, we modified both layers of U-Boot, as well as the Linux kernel, to incorporate the generated entropy into the kernel randomness pools. We use a custom extracting hash to condense the memory decay into 1,024 bits, and pass the result into the kernel as a base-64-encoded parameter. Overall, hashing and processing takes less than second, on top of the unavoidable multiple-second DRAM decay time.

4.4.4 Results

Decay Probability and Distribution

Although we could reliably observe many bits decaying, the distribution of decay was not uniform. Figure 4.15 shows the distribution of decay probabilities at 58 seconds. The values range from 0 (white) to very low (green) to near certainty (red). The figure also shows that some areas of the DRAM do not appear to decay at all.
The horizontal bands in the figure are due to the test pattern initially written to memory. We wrote 0xAA to the top quarter of memory, 0x00 to the next quarter, 0xFF to the next, and 0x55 to the last quarter. In the areas which show no decay, the pattern (0x00 or 0xFF) matched the cell’s “ground state” (i.e., the state into which the cell naturally decays). This can vary because chips use different voltages for “0” and “1” in different portions of the chip.

Figure 4.13 shows decay over time. The yellow bits decayed first and the red bits decayed last. Unsurprisingly, the longer the interval, the more bit errors occur and the more randomness we are able to extract. In Figure 4.14, each bit’s probability of decay over 7, 14, and 28 seconds has been graphed. Perhaps unsurprisingly, every bit that ever decays within 7 seconds has a 100% chance of decaying in 14 or 28 seconds. Interestingly, a number of bits with a non-zero probability of decaying in 14 seconds don’t always decay by 28 seconds, indicating that DRAM bits don’t simply decay in a set order, and can provide true entropy.

**Temperature Dependence**

Previous work has shown that decay varies with DRAM temperature. To compensate, many systems increase the refresh frequency as temperature increases [75, 133]. Primarily, this is due to the increase in DRAM cell leakage as temperature rises [75]. To understand the effect of this temperature dependence on the probability of decay, we set up an experimental protocol that allowed us to control the temperature of DRAM. By submerging DRAM in non-conductive isopropyl alcohol inside a refrigerator and using an aquarium heater, we were able to control the DRAM temperature to ±1°C. For temperatures above 35°C, we used a laboratory oven to measure decay at high temperatures.
Our results are shown in figure 4.16. We find that at low temperatures, the few bits which decay are consistent (i.e. the same bits always decay). Around 20° C, we begin to see bits that sometimes decay. At room temperature (23° C), we begin to see an exponential rise in bit decay.

**DRAM Temperature Modification**

We can generate more randomness by increasing the temperature of the DRAM chip. To accomplish this we wrote a simple ‘power virus’ that attempts to maximize DRAM power consumption and heat dissipation. The virus initializes a region of DRAM and then reads repeatedly from different addresses. The initial data and the addresses are chosen to maximize the transition frequency on the DRAM pins.

We find that by implementing our power virus, we heat up the DRAM from 26° C to 29° C within 1 minute. We run the power virus while waiting for bits to decay.
Variability

In addition to randomness in bit decay between boots, we also observed two kinds of variability between individual boards: Decay probability variability, the variability in the probability that different bits will decay; and cold state variability, the variability in the initial contents of DRAM after a cold boot.

This variability is due to manufacturing variations that cause DRAM cells to leak at different rates, leading to the decay probability variability we observe. Process variations in the sense amplifiers (which convert the analog values from the DRAM bits into logical “0”s and “1”s) is also well documented [58], and probably contributes as well.

The variation in the DRAM’s contents from a cold boot (measured after the device was powered off for 3 days) can provide a unique fingerprint for each board. For instance, at 25–28 °C with a delay of 7s, on one BeagleBoard, a certain 10 bits always decay, while the other BeagleBoard has only 6 such bits. The two sets are disjoint; that is, the bits that decay on one board do not decay on the other.

Under the assumption that, due to process variation, the placement of these “leaky” bits is independent between different DRAM modules, the locations of leaky bits act as a fingerprint for a particular BeagleBoard. Verifying this assumption about the distribution of leaky bits would require access to more systems than we have, and we leave it to future work.

4.4.5 Extracting per-boot randomness from DRAM

The location of leaky bits cannot, by itself, be the basis for random number generation. An attacker who has physical access, who can mount a remote code-injection exploit, or can otherwise run software on the device will be able to locate its leaky bits. Therefore, we must examine the bits that sometimes decay and sometimes do not.
We now give a rough estimate for the amount of entropy available in the decay of these bits. Our analysis makes the assumption that bits decay independently of each other. This is a strong assumption and there evidence that it is at least partly false, e.g., Section 3.3 of [56]. Accordingly, the entropy estimates below are overestimates. We hope future work can provide a better measure of available entropy.

We estimate $Pr[\text{decay}]$ for each bit based on our experiments and use this probability to compute the information theoretic entropy content for this bit:

$$E(p) = - (p \cdot \log_2(p) + (1 - p) \cdot \log_2(1 - p))$$  \hspace{1cm} (4.3)

Under the assumption that bits decay independently of each other, we can simply sum this distribution entropy over every bit we saw decay.

For a BeagleBoard xM at 25-27°C and with a decay time of 7 s, we obtain a total boot-time entropy estimate of 4.9 bits, largely due to the fact that only 19 memory decays ever happen, and 16 of these happen with $p > 0.9$ or $p < 0.1$. For a decay time of 14s, we see 511 bits ever decay, and summing their entropy contributions gives an entropy estimate of 209.1 bits. For a delay of 28s, 9,943 bits decay, for an estimated entropy of 8,415.16 bits. For 56 seconds, we see 427,062 decays, for an estimated entropy of 98,611.85 bits.

A delay of even 14s on first boot is unacceptable in many applications. Moreover, because DRAM decay depends on temperature, this approach may not provide security in very cold conditions — for example, phones used on a ski slope.

### 4.5 PLL Lock Latency

The PLLs that produce the on-chip clocks in modern processors are complex, analog devices. When they start up (or the chip reconfigures them), they take a variable
amount of time to “lock” on to the new output frequency (see 3 in Figure 4.11). This variation in lock time is due to a number of factors, including stability of the power supply, accuracy and jitter in the source oscillator, temperature, and manufacturing process variation [60]. Repeatedly reconfiguring an on-chip PLL and measuring how long it takes to lock will result in random variations.

SoCs typically contain several PLLs used to derive clocks for the processor, memory and peripherals. On the BeagleBoard xM, the DM3730 contains 5 DPLLs (Digital Phase Locked Loops). Each DPLL can be reconfigured and toggled via a software register, and a status flag and interrupt will signal when a DPLL is locked. To measure the time it takes to acquire a lock, we instrumented code to disable the DPLL for the USB peripheral clock on the BeagleBoard xM. Using the hardware performance counter, we measured the number of cycles it took for the DPLL to reacquire a lock (Figure 4.17).

We obtain about 4.7 bits of entropy every time we re-lock the DPLL, and it takes at most approximately 9000 cycles (9 µs) for the DPLL to re-lock. Using the DPLL lock latency, we can obtain about 522 KiB of pure entropy per second.

DPLL lock latency could be easily polled for entropy during early boot when the SoC first sets up the clocks and PLLs in the system. Since the DPLL is affected by analog conditions such as temperature, a determined attacker may be able to induce bias in the lock time.

4.6 Conclusions

Randomness is a fundamental system service. A system cannot be said to have successfully booted unless it is ready to provide high-entropy randomness to applications.

We have presented three techniques for gathering entropy early in the boot process. These techniques provide different tradeoffs along three metrics: how high the bitrate,
how specific to a particular system, and how well explained by unpredictable physical processes.

Our first technique, which times the execution of kernel code blocks, provides a moderate amount of entropy and is easily applied to every system we examined, but we are able to give only a partial account for the source of the entropy it gathers.

Our second technique, DRAM decay, provides a large amount of entropy, but presents a heavy performance penalty and is tricky to deploy, relying on details of the memory controller. Its advantage is a physical justification for the observed randomness.

Our third technique, timing PLL locking, promises the highest bitrate and is well supported by physical processes, but its implementation requires intimate knowledge of the individual SoC.

We implemented and characterized these techniques on a broad spectrum of embedded devices featuring a variety of popular SoCs and hardware, from resource-rich mobile phone hardware to devices that aren’t much more than an ethernet port and a SoC. While these three techniques certainly can be applied to traditional desktop systems as
well as more powerful embedded devices, in some sense, our tiny embedded systems start at a disadvantage. Wireless devices can read randomness from radios; desktops can rely on saved entropy from previous boots. Our work focuses on adequately protecting headless, resource-poor embedded devices, which must acquire strong entropy on their very first boot, before they can even export network connectivity.

Our work leaves many questions open. We are able to give only a partial explanation for the entropy we observed in our first technique, and only a partial characterization of the DRAM decay effects in our second technique. We hope that future work can shed more light on the situation. More work is also necessary to understand how much the gathered entropy depends on environmental factors that might be under adversarial control.

The three techniques we present exploit just a few of the many potential architectural sources of randomness available in modern systems. Other possible sources of entropy, which we hope will be explored in future work, include voltage scaling latency, GPIO pin voltage, flash memory corruption patterns, and power supply stabilization latency.

Our three techniques are all, ultimately, workarounds for the lack of dedicated hardware random number generators in embedded architectures. What will spur the adoption of such hardware, by both hardware and software developers? What is the right way to specify such hardware for the ARM architecture, where a high-level core description is licensed to many processor manufacturers? Furthermore, is it possible to verify that such a unit is functioning correctly and free of backdoors?

Acknowledgments

We thank Daniel J. Bernstein, J. Alex Halderman, Nadia Heninger, and Eric Rescorla for their comments and suggestions.
We would like to thank Xilinx for donating hardware.

This material is based upon work supported by the National Science Foundation under Grants No. CNS-0831532, CNS-0964702, DGE-1144086, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

Chapter 4, in part, is a reprint of the material as it appears in IEEE Security and Privacy (Oakland) 2013. Mowery, Keaton; Wei, Michael; Kohlbrenner, David; Shacham, Hovav; Swanson, Steven, 2013. The dissertation author was a primary investigator and the primary author of this paper.
Chapter 5

Security Analysis of a Full-Body Scanner

Advanced imaging technologies are a new class of people screening systems used at airports and other sensitive environments to detect metallic as well as nonmetallic contraband. We present the first independent security evaluation of such a system, the Rapiscan Secure 1000 full-body scanner, which was widely deployed at airport checkpoints in the U.S. from 2009 until 2013. We find that the system provides weak protection against adaptive adversaries: It is possible to conceal knives, guns, and explosives from detection by exploiting properties of the device’s backscatter X-ray technology. We also investigate cyberphysical threats and propose novel attacks that use malicious software and hardware to compromise the the effectiveness, safety, and privacy of the device. Overall, our findings paint a mixed picture of the Secure 1000 that carries lessons for the design, evaluation, and operation of advanced imaging technologies, for the ongoing public debate concerning their use, and for cyberphysical security more broadly.
5.1 Introduction

In response to evolving terrorist threats, including non-metallic explosive devices and weapons, the U.S. Transportation Security Administration (TSA) has adopted advanced imaging technology (AIT), also known as whole-body imaging, as the primary passenger screening method at nearly 160 airports nationwide [140]. Introduced in 2009 and gradually deployed at a cost exceeding $1 billion, AIT provides, according to the TSA, “the best opportunity to detect metallic and non-metallic anomalies concealed under clothing without the need to touch the passenger” [137].

AIT plays a critical role in transportation security, and decisions about its use are a matter of public interest. The technology has generated considerable controversy, including claims that the devices are unsafe [124], violate privacy and civil liberties [126, 92], and are ineffective [82, 32]. Furthermore, AIT devices are complex cyberphysical systems — much like cars [87] and implantable medical devices [50] — that raise novel computer security issues. Despite such concerns, neither the manufacturers nor the government agencies that deploy these machines have disclosed sufficient technical details to facilitate rigorous independent evaluation [124], on the grounds that such information could benefit attackers [137]. This lack of transparency has limited the ability of policymakers, experts, and the public to assess contradicting claims.

To help advance the public debate, we present the first experimental analysis of an AIT conducted independently of the manufacturer and its customers. We obtained a Rapiscan Secure 1000 full-body scanner — one of two AITs widely deployed by the TSA [108] — and performed a detailed security evaluation of its hardware and software. Our analysis provides both retrospective insights into the adequacy of the testing and evaluation procedures that led to TSA use of the system, and prospective lessons about broader security concerns, including cyberphysical threats, that apply to current and future AITs.
The Rapiscan Secure 1000 full-body scanner uses backscattered X-rays to construct an image through clothing. Naïvely hidden contraband, such as the handgun tucked into this subject’s waistband, is readily visible to the device operator.

The Secure 1000 provides a unique opportunity to investigate the security implications of AITs in a manner that allows robust yet responsible public disclosure. Although it was used by the TSA from 2009 until 2013, it has recently been removed from U.S. airports due to changing functional requirements [112]. Moreover, while the Secure 1000 uses backscatter X-ray imaging, current TSA systems are based on a different technology, millimeter waves [38], so many of the attacks we present are not directly applicable to current TSA checkpoints, thus reducing the risk that our technical disclosures will inadvertently facilitate mass terrorism. However, while Secure 1000 units are no longer used in airports, they still are in use at other government facilities, such as courthouses and prisons (see, e.g., [63, 97]). In addition, other backscatter X-ray devices manufactured by American Science and Engineering are currently under consideration for use at airports [112]. To mitigate any residual risk, we have redacted a small number of sensitive details from our attacks in order to avoid providing recipes that would allow an
attacker to reliably defeat the screening process without having access to a machine for testing.

In the first part of our study (Section 5.3), we test the Secure 1000’s effectiveness as a physical security system by experimenting with different methods of concealing contraband. While the device performs well against naïve adversaries, fundamental limitations of backscatter imaging allow more clever attackers to defeat it. We show that an adaptive adversary, with the ability to refine his techniques based on experiment, can confidently smuggle contraband past the scanner by carefully arranging it on his body, obscuring it with other materials, or properly shaping it. Using these techniques, we are able to hide firearms, knives, plastic explosive simulants, and detonators in our tests. These attacks are surprisingly robust, and they suggest a failure on the part of the Secure 1000’s designers and the TSA to adequately anticipate adaptive attackers. Fortunately, there are simple procedural changes that can reduce (though not eliminate) these threats, such as performing supplemental scans from the sides or additional screening with a magnetometer.

Next, we evaluate the security of the Secure 1000 as a cyberphysical system (Section 5.4) and experiment with three novel kinds of attacks against AITs that target their effectiveness, safety features, and privacy protections. We demonstrate how malware infecting the operator’s console could selectively render contraband invisible upon receiving a “secret knock” from the attacker. We also attempt (with limited success) to use software-based attacks to bypass the scanner’s safety interlocks and deliver an elevated radiation dose. Lastly, we show how an external device carried by the attacker with no access to the console can exploit a physical side-channel to capture naked images of the subject being scanned. These attacks are, in general, less practical than the techniques we demonstrate for hiding contraband, and their limitations highlight a series of conservative engineering choices by the system designers that should serve as positive examples for future AITs.
Finally, we attempt to draw broader lessons from these findings (Section 5.5). Our results suggest that while the Secure 1000 is effective against naïve attackers, it is not able to guarantee either efficacy or privacy when subject to attack by an attacker who is knowledgeable about its inner workings. While some of the detailed issues we describe are specific to the scanner model we tested, the root cause seems to be the failure of the system designers and deployers to think adversarially. This pattern is familiar to security researchers: past studies of voting machines [21], cars [87] and medical devices [50] have all revealed cyberphysical systems that functioned well under normal circumstances but were not secure in the face of attack. Thus, we believe this study reinforces the message that security systems must be subject to adversarial testing before they can be deemed adequate for widespread deployment.

**Research safety and ethics**

Since the Secure 1000 emits ionizing radiation, it poses a potential danger to the health of scan subjects, researchers, and passers by. Our institutional review board determined that our study did not require IRB approval; however, we worked closely with research affairs and radiation safety staff at the university that hosted our device to minimize any dangers and assure regulatory compliance. To protect passers by, our device was sited in a locked lab, far from the hallway, and facing a thick concrete wall. To protect researchers, we marked a 2 m region around the machine with tape; no one except the scan subject was allowed inside this region while high voltage was applied to the X-ray tube. We obtained a RANDO torso phantom [111], made from a material radiologically equivalent to soft tissue cast over a human skeleton, and used it in place of a human subject for all but the final confirmatory scans. For these final scans we decided, through consultation with our IRB, that only a PI would be used as a scan subject. Experiments involving weapons were conducted with university approval and in coordination with the
campus police department and all firearms were unloaded and disabled. We disclosed our security-relevant findings and suggested procedural mitigations to Rapiscan and the Department of Homeland Security ahead of publication.

Online material

Additional resources and the most recent version of this report are available online at https://radsec.org/.

5.2 The Rapiscan Secure 1000

The Secure 1000 was initially developed in the early 1990s by inventor Steven W. Smith [129, 127]. In 1997, Rapiscan Systems acquired the technology [128] and began to produce the Rapiscan Secure 1000. In 2007, the TSA signed a contract with Rapiscan to procure a customized version of the Secure 1000 for deployment in airport passenger screening [136].

We purchased a Rapiscan Secure 1000 from an eBay seller who had acquired it in 2012 at a surplus auction from a U.S. Government facility located in Europe [68]. The system was in unused condition. It came with operating and maintenance manuals as well as detailed schematics, which were a significant aid to reverse engineering. The system consists of two separate components: the scanner unit, a large enclosure that handles X-ray generation and detection under the control of a special purpose embedded system, and the user console, a freestanding cabinet that contains a PC with a keyboard and screen. The two components are connected by a 12 m cable.

The system we tested is a dual pose model, which means that the subject must turn around in order to be scanned from the front and back in two passes. TSA screening checkpoints used the Secure 1000 single pose model [108], which avoids this inconvenience by scanning from the front and back using a pair of scanner units. Our system
was manufactured in about September 2006 and includes EPROM software version 2.1. Documents obtained under the Freedom of Information Act suggest that more recent versions of the hardware and software were used for airport screening [142, 131], and we highlight some of the known differences below. Consequently, we focus our analysis on fundamental weaknesses in the Secure 1000 design that we suspect also affect newer versions. A detailed analysis of TSA models might reveal additional vulnerabilities.

5.2.1 Backscatter Imaging

X-ray backscatter imaging exploits the unique properties of ionizing radiation to penetrate visual concealment and detect hidden contraband. The physical process which generates backscatter is Compton scattering, in which a photon interacts with a loosely bound or free electron and scatters in an unpredictable direction [31]. Other interactions, such as the photoelectric effect, are possible, and the fraction of photons that interact and which particular effect occurs depends on each photon’s energy and the atomic composition of the mass. For a single-element material, the determining factor is its atomic number \( Z \), while a compound material can be modeled by producing an “effective \( Z \),” or \( Z_{\text{eff}} \) [132].

Under constant-spectrum X-ray illumination, the backscattered intensity of a given point is largely determined by the atomic composition of matter at that location, and to a lesser extent its density. Thus, organic materials, like flesh, can be easily differentiated from materials such as steel or aluminum that are made from heavier elements.

The Secure 1000 harnesses these effects for contraband screening by operating as a “reverse camera,” as illustrated in Figure 5.2. X-ray output from a centrally-located tube (operating at 50 kVp and 5 mA) passes through slits in shielding material: a fixed horizontal slit directly in front of a “chopper wheel,” a rapidly spinning disk with four
Figure 5.2. Backscatter Imaging—An X-ray tube (A) mounted on a platform travels vertically within the scanner. The X-rays pass through a spinning disk (B) that shapes them into a horizontally scanning beam. Some photons that strike the target (C) are backscattered toward detectors (D) that measure the reflected energy over time. Adapted from U.S. Patent 8,199,996 [65].

radial slits. This results in a narrow, collimated X-ray beam, repeatedly sweeping across the imaging field. During a scan, which takes about 5.7 s, the entire X-ray assembly moves vertically within the cabinet, such that the beam passes over every point of the scene in a series of scan lines.

As the beam sweeps across the scene, a set of 8 large X-ray detectors measures the intensity of the backscattered radiation at each point, by means of internal photomultiplier tubes (PMTs). The Secure 1000 combines the output of all 8 detectors, and sends the resulting image signal to the user console, which converts the time-varying signal into a $160 \times 480$ pixel monochrome image, with the intensity of each pixel determined by the $Z_{eff}$ value of the surface of the scan subject represented by that pixel location.
5.2.2 Subsystems

Operator interface

The operator interacts with the Secure 1000 through the user console, a commodity x86 PC housed within a lockable metal cabinet. With our system, the user console is connected to the scanner unit via a serial link and an analog data cable. Documents released by the TSA indicate that airport checkpoint models were configured differently, with an embedded PC inside the scanner unit linked to a remote operator workstation via a dedicated Ethernet network [131, 142].

On our unit, the operator software is an MS-DOS application (SECURE65.EXE) that launches automatically when the console boots. (TSA models are apparently Windows-based and use different operator software [136, 131].) This software is written in a BASIC variant, and the main user interface is a $640 \times 480$ pixel, 4-bit grayscale
screen, as shown in Figure 5.3. The operator invokes a scan by pressing a hand switch. After image acquisition, the operator can inspect the scan by means of a 2× zoom and interactive brightness and contrast controls. The image can also be saved to disk or printed. Further, the software contains several calibration functions that can only be accessed by entering a 4 digit numeric password. The password is hard-coded and is printed in the maintenance manual.

Scanner unit

The scanner unit contains an assortment of electrical and mechanical systems under the control of an embedded computer called the System Control Board (SCB). The SCB houses an Intel N80C196KB12 microcontroller, executing software contained on a 32 KiB socketed ROM. It interacts with the user console PC over a bidirectional RS-232 serial link using simple ASCII commands such as SU for “scan up” and SD for “scan down.” In turn, the SCB uses digital and analog interfaces to direct and monitor other components, including the X-ray tube, PMTs, and chopper wheel. It also implements hardware-based safety interlocks on the production of X-rays, which we discuss further in Section 5.4.2.

To control vertical movement of the X-ray tube, the scanner unit uses an off-the-shelf reprogrammable servo motor controller, the Parker Gemini GV6. In normal operation, the servo controller allows the SCB to trigger a movement of the X-ray tube, initially to a “home” position and subsequently to scan up and down at predefined rates. There is no command to move the tube to a specific intermediate position.
5.3 Contraband Detection

As the Secure 1000 is intended to detect prohibited or dangerous items concealed on the body of an attacker, the first and most obvious question to ask is how effectively the Secure 1000 detects contraband.

To make the discussion concrete, we consider the machine as it was typically used by the TSA for airport passenger screening. Under TSA procedures, subjects were imaged from the front and back, but not from the sides. A trained operator inspected the images and, if an anomaly was detected, the passenger was given a manual pat down to determine whether it was a threat [131]. The Secure 1000 was used in place of a walk-through metal detector, rather than both screening methods being employed sequentially [137]. We focus our analysis on threats relevant to an airport security context, such as weapons and explosives, as opposed to other contraband such as illicit drugs or bulk currency.

To replicate a realistic screening environment, we situated our Secure 1000 in an open area, oriented 2.5 m from a concrete wall sufficient to backstop X-ray radiation. This distance accords with the manufacturer’s recommendation of at least 2 m of open area “for producing the best possible images” [115]. For typical tests, we arranged the subject at a distance of about 38 cm in front of the scanner using the foot position template provided with the machine.

Naïve adversary

First, we consider the scanner’s effectiveness against a naïve adversary, an attacker whose tactics do not change in response to the introduction of the device. Although this is a weak attacker, it seems to correspond to the threat model under which the scanner was first tested by the government, in a 1991 study of a prototype of the Secure 1000 conducted by Sandia National Laboratories [85]. Our results under this threat model
generally comport with theirs. Guns, knives, and blocks of explosives naïvely carried on the front or back of the subject’s body are visible to the scanner operator.

Three effects contribute to the detectability of contraband. The first is contrast: human skin appears white as it backscatters most incident X-ray radiation, while metals, ceramics, and bone absorb X-rays and so appear dark gray or black. The second is shadows cast by three-dimensional objects as they block the X-ray beam, which accentuate their edges. The third is distortion of the subject’s flesh as a result of the weight of the contraband or the mechanics of its attachment. The naïve adversary is unlikely to avoid all three effects by chance.

A successful detection of hidden contraband can be seen in Figure 5.1. The subject has concealed a .380 ACP pistol within his waistband. The X-ray beam interacts with the gun metal significantly differently than the surrounding flesh, and the sharp contrast in backscatter intensity is immediately noticeable.

**Adaptive adversary**

Of course, real attackers are not entirely ignorant of the scanner. The TSA announced that it would be used at screening checkpoints [137, 44], the backscatter imaging mechanism is documented in patents and manufacturer reports [65, 88, 114], images captured with the device have appeared in the media [44, 89], and the physics of backscatter X-rays are well understood [31, 6, 85]. We must assume that attackers have such information and adapt their tactics in response.

To simulate an adaptive adversary, we performed experiments in the style of white-box penetration testing commonly employed in the computer security field. We allowed ourselves complete knowledge of how the scanner operates as well as the ability to perform test scans, observed the resulting images, and used them to adjust our concealment methods.
(a) Subject with .380 ACP pistol taped above knee.  (b) Subject with .380 ACP pistol sewn to pant leg.

**Figure 5.4.** Concealing a Pistol by Positioning — The Secure 1000 cannot distinguish between high $Z_{eff}$ materials, such as a metal handgun, and the absence of a backscatter response. Carefully placed metallic objects can be invisible against the dark background.

Such interactive testing is not strictly necessary to develop clever attacks. Indeed, researchers with no access to the Secure 1000 have proposed a number of concealment strategies based only on published information [82], and we experimentally confirm that several of these attacks are viable. However, the ability to perform tests substantially increases the probability that an attack will succeed on the first attempt against a real deployment. A determined adversary might acquire this level of access in several ways: by buying a machine, as we did; by colluding with a dishonest operator; or by probing the security of real installations over time.
In the remainder of this section, we describe experiments with three adaptive concealment techniques and show that they can be used to defeat the Secure 1000. We successfully use them to smuggle firearms, knives, and explosive simulants past the scanner.

### 5.3.1 Concealment by Positioning

The first concealment technique makes use of a crucial observation about X-ray physics: backscatter screening machines emitting X-rays in the 50 keV range, such as the Secure 1000, cannot differentiate between the absence of matter and the existence of materials with high $Z_{\text{eff}}$ (e.g., iron and lead). That is, when the scanner emits probing X-rays in a direction and receives no backscatter, it can either be because the beam interacted with nothing, i.e., traveled unimpeded past the screening subject, or because the beam shone directly upon a material which absorbed it entirely and thus did not backscatter. In either case, the resulting pixels will be dark.

These facts lead directly to a straightforward concealment attack for high $Z_{\text{eff}}$ contraband: position the object such that it avoids occluding the carrier’s body with respect to the X-ray beam. This technique was first suggested on theoretical grounds by Kaufman and Carlson [82]. In limited trials, a TSA critic used it to smuggle small metal objects through airport checkpoints equipped with the Secure 1000 and other AITs [32]. Note that this attack is not enabled by a poor choice of image background color; as discussed above, the scanner cannot differentiate between the metal objects and the absence of material.

To more fully investigate this attack, we obtained a set of weapons: both knives and firearms, ranging from a .380 ACP pistol to an AR-15 semi-automatic rifle. When we scanned the weapons against a dark backdrop, most of the firearms were readily visible due to the presence of nonmetallic parts. After testing a number of firearms, we settled on our .380 ACP pistol as the most suitable candidate for concealment.
No contraband

18 cm knife taped to spine

Knife behind 1.5 cm plastic block

Figure 5.5. Concealing a Knife by Masking—We find that high-$Z_{\text{eff}}$ materials can be hidden by covering them with lower $Z_{\text{eff}}$ materials, such as the common plastic PTFE (Teflon). For example, a metal knife is clearly visible when naively concealed, but when covered with a thin plastic block it approximates the color of the spine. Tapering the block’s edges would reduce the visible outline.

We performed several trials to test different placement and attachment strategies. In the end, we achieved excellent results with two approaches: carefully affixing the pistol to the outside of the leg just above the knee using tape, and sewing it inside the pant leg near the same location. Front and back scans for both methods are shown in Figure 5.4. In each case, the pistol is invisible against the dark background, and the attachment method leaves no other indication of the weapon’s presence.

In a similar test, we concealed an 11 cm metal folding knife, in its closed position, along our test subject’s side. In this case, too, front and back scans were completely unable to detect the weapon.

Fortunately, simple procedural changes can thwart these attacks. Instead of performing only front and back scans, every subject could also be made to undergo scans from the left and right sides. Under these scans, a high $Z_{\text{eff}}$ weapon positioned on the side of the body would be as obvious as the one in Figure 5.1. Unfortunately, these
additional scans would nearly halve the maximum throughput of the checkpoint, as well as double each person’s radiation dose. Another possible mitigation would be to screen each subject with a magnetometer, which would unequivocally find metallic contraband but would fail to uncover more exotic weapons, such as ceramic knives [140, 144]. We note that the attacker’s gait or appearance might be compromised by the mass and bulk of the firearm or knife, and this might be noticeable to security personnel outside of the backscatter X-ray screening.

5.3.2 Concealment by Masking

The second object concealment techniques we attempted are similarly based on X-ray physics: the brightness of a material in the image is directly correlated to its backscatter intensity, which in turn is determined by the $Z_{\text{eff}}$ and density of the matter in the path of the beam. Therefore, any combination of substances which scatter incoming X-rays at the same approximate intensity as human flesh will be indistinguishable from the rest of the human.

One consequence of this fact is that high-$Z_{\text{eff}}$ contraband can be concealed by masking it with an appropriate thickness of low-$Z_{\text{eff}}$ material. We experimented with several masking materials to find one with a $Z_{\text{eff}}$ value close to that of flesh. We obtained good results with the common plastic PTFE (Teflon), although due to its low density a significant thickness is required to completely mask a metallic object.

To work around this issue, we took advantage of the Secure 1000’s ability to see bones close to the skin. Figure 5.5 demonstrates this approach: an 18 cm knife is affixed to the spine and covered with 1.5 cm of PTFE. As the X-rays penetrate through the material, they backscatter so that the knife outline approximates our subject’s spine. While this mask arrangement creates hard edges and shadows which render it noticeable to screening personnel these effects could be reduced by tapering the edges of the mask.
A more difficult challenge for the attacker is taking into account the anatomy of the specific person being imaged. Shallow bones and other dense tissue are visible to the scanner under normal conditions, and a poorly configured mask will stand out against these darker areas of the scan. We conclude that masking can be an effective concealment technique, but achieving high confidence of success would require access to a scanner for testing.

5.3.3 Concealment by Shaping

Our third and final concealment technique applies a strategy first theorized in [82] to hide malleable, low-\(Z_{\text{eff}}\) contraband, such as plastic explosives. These materials produce low contrast against human flesh, and, unlike rigid weapons, the attacker can reshape them so that they match the contours of the body.

To experiment with this technique, we acquired radiological simulants for both Composition C-4 [149] and Semtex [150], two common plastic high explosives. These simulants are designed to emulate the plastic explosives with respect to X-ray interactions, and both are composed of moldable putty, similar to the actual explosive materials. We imaged both C-4 and Semtex simulants with the Secure 1000, and found that they appear very similar. We selected the C-4 simulant for subsequent tests.

Our initial plan was to modify the simulants’ \(Z_{\text{eff}}\) to better match that of flesh, by thoroughly mixing in fine metallic powder. To our surprise, however, a thin pancake (about 1 cm) of unmodified C-4 simulant almost perfectly approximated the backscatter intensity of our subject’s abdomen.

We affixed the pancake with tape (which is invisible to the Secure 1000), and faced two further problems. First, the pancake covered our subject’s navel, which is normally clearly visible as a small black area in the scans. Second, by design, plastic explosives are almost completely inert without a matching detonator. These problems
Figure 5.6. Concealing Explosives by Shaping — *Left:* Subject with no contraband. *Right:* Subject with more than 200 g of C-4 plastic explosive simulant plus detonator, molded to stomach.
neatly solve each other: we attached a detonator, consisting of a small explosive charge in a metal shell, directly over our subject’s navel. Since the detonator is coated in metal, it absorbs X-rays quite well and mimics the look of the navel in the final image.

Figure 5.6 shows a side-by-side comparison of our test subject both carrying no contraband and carrying 200 g of C-4 explosive and attached detonator. To put this amount in perspective, “Shoe Bomber” Richard Reid reportedly carried about 280 g of explosive material [30], and the bomb that destroyed Pan Am Flight 103 is thought to have contained 350 g of Semtex [147].

These scans indicate that plastic explosives can be smuggled through a Secure 1000 screening, since thin pancakes of these materials do not contrast strongly with flesh. While a metal detector would have been sufficient to detect the detonator we used, not all detonators have significant metal components.

In summary, an adaptive adversary can use several attack techniques to carry knives, guns, and plastic explosives past the Secure 1000. However, we also find that multiple iterations of experimentation and adjustment are likely necessary to achieve consistent success. The security of the Secure 1000, then, rests strongly on the adversary’s inability to acquire access to thedevice for testing. However, since we were able to purchase a Secure 1000, it is reasonable to assume that determined attackers and well-financed terrorist groups can do so as well. We emphasize that procedural changes — specifically, performing side scans and supplementing the scanner with a magnetometer — would defeat some, though not all, of the demonstrated attacks.

### 5.4 Cyberphysical Attacks

The Secure 1000, like other AITs, is a complex cyberphysical system. It ties together X-ray emitters, detectors, and analog circuitry under the control of embedded
Figure 5.7. A Secret Knock—We demonstrate how malware infecting the Secure 1000 user console could be used to defeat the scanner. The malware is triggered when it detects a specific pattern in a scan, as shown here. It then replaces the real image (c) of the attacker, which might reveal hidden contraband, with an innocuous image stored on disk. Pattern recognition occurs in real time.

computer systems, and feeds the resulting image data to a traditional desktop system in the user console. In this section, we investigate computer security threats against AITs. We demonstrate a series of novel software- and hardware-based attacks that undermine the Secure 1000’s efficacy, safety features, and privacy protections.

5.4.1 User Console Malware

The first threat we consider is malware infecting the user console. On our version of the Secure 1000, the user console is an MS-DOS–based PC attached to the scanner unit via a proprietary cable; TSA models apparently used Windows and a dedicated Ethernet switch [136, 138]. Although neither configuration is connected to an external network, there are several possible infection vectors. If the operators or maintenance personnel are malicious, they could abuse their access in order to manually install malware. The software on our machine lacks any sort of electronic access controls (e.g., passwords) or software verification. While the PC is mounted in a lockable cabinet, we were able to
pick the lock in under 10 seconds with a commercially available tool. Therefore, even an outsider with temporary physical access could easily introduce malicious code. TSA systems may be better locked down, but sophisticated adversaries have a track record of infecting even highly secured, airgapped systems [91, 105].

We implemented a form of user console malware by reverse engineering SECURE65.EXE, the front-end software package used by the Secure 1000, and creating a malicious clone. Our version, INSECURE.EXE, is a functional, pixel-accurate reimplementation of the original program and required approximately one man-month to create.

In addition to enabling basic scanning operations, INSECURE.EXE has two malicious features. First, every scan image is saved to a hidden location on disk for later exfiltration. This is a straightforward attack, and it demonstrates one of many ways that software-based privacy protections can be bypassed. Of course, the user could also take a picture of the screen using a camera or smartphone — although operators are forbidden to have such devices in the screening room [121].

Second, INSECURE.EXE selectively subverts the scanner’s ability to detect contraband. Before displaying each scan, it applies a pattern recognition algorithm to look for a “secret knock” from the attacker: the concentric squares of a QR code position block. If this pattern occurs, INSECURE.EXE replaces the real scan with a preprogrammed innocuous image. The actual scan, containing the trigger pattern and any other concealed contraband, is entirely hidden.

To trigger this malicious substitution, the subject simply wears the appropriate pattern, made out of any material with a sufficiently different $Z_{eff}$ than human tissue. In our experiments, we arranged lead tape in the target shape, attached to an undershirt, as shown in Figure 5.7. When worn under other clothing, the target is easily detected by the malware but hidden from visual inspection.
Recently, in response to privacy concerns, the TSA has replaced manual review of images with algorithmic image analysis software known as automated target recognition (ATR) [141]. Instead of displaying an image of the subject, this software displays a stylized figure, with graphical indicators showing any regions which the software considers suspect and needing manual resolution. (Delays in implementing this algorithm led the TSA to remove Secure 1000 machines from airports entirely [5].) If malware can compromise the ATR software or its output path, it can simply suppress these indicators — no image replacement needed.

### 5.4.2 Embedded Controller Attacks

The System Control Board (SCB) managing the physical scanner is a second possible point of attack. While the SCB lacks direct control over scan images, it does control the scanner’s mechanical systems and X-ray tube. We investigated whether an attacker who subverts the SCB firmware could cause the Secure 1000 to deliver an elevated radiation dose to the scan subject.

This attack is complicated by the fact that the Secure 1000 includes a variety of safety interlocks that prevent operation under unexpected conditions. Circuits sense removal of the front panel, continuous motion of the chopper wheel and the vertical displacement servo, X-ray tube temperature and supply voltage, X-ray production level, key position (“Standby” vs. “On”), and the duration of the scan, among other parameters. If any anomalous state is detected, power to the X-ray tube is immediately disabled, ceasing X-ray emission.

While some of these sensors merely provide inputs to the SCB software, others are tied to hard-wired watchdog circuits that cut off X-ray power without software mediation. However, the firmware can bypass these hardware interlocks. At the beginning of each scan, operational characteristics such as tube voltage and servo motion fluctuate outside
their nominal ranges. To prevent immediate termination of every scan, SCB software temporarily asserts a bypass signal, which disables the hardware interlocks. This signal feeds a “bypass watchdog” circuit of its own, meant to prevent continual interlock bypass, but the SCB can pet this watchdog by continuously toggling the bypass signal, and cause all hardware interlocks to be ignored. Thus, every safety interlock is either directly under software control or can be bypassed by software.

We developed replacement SCB firmware capable of disabling all of the software and hardware safety interlocks in the Secure 1000. With the interlocks disabled, corrupt firmware can, for instance, move the X-ray tube to a specific height, stop the chopper wheel, and activate X-ray power, causing the machine to deliver the radiation dose from an entire dose to a single point. Only the horizontal displacement of this point is not directly under firmware control — it depends on where the chopper wheel happens to come to rest.

Delivering malicious SCB firmware presents an additional challenge. The firmware is stored on a replaceable socketed EPROM inside the scanner unit, which is secured by an easily picked wafer tumbler lock. Although attackers with physical access could swap out the chip, they could cause greater harm by, say, hiding a bomb inside the scanner. For SCB attacks to pose a realistic safety threat, they would need to be remotely deployable.

Due to the scanner’s modular design, the only feasible vector for remote code execution is the serial link between the user console and the SCB. We reverse engineered the SCB firmware and extensively searched for vulnerabilities. The firmware is simple (<32 KiB) and appears to withstand attacks quite well. Input parsing uses a fixed length buffer, to which bytes are written from only one function. This function implements bounds checking correctly. Data in the buffer is always processed in place, rather than being copied to other locations that might result in memory corruption. We were unable to cause any of this code to malfunction in a vulnerable manner.
While we are unable to remotely exploit the SCB to deliver an elevated radiation dose, the margin of safety by which this attack fails is not reassuring. Hardware interlocks that can be bypassed from software represent a safety mechanism but not a security defense. Ultimately, the Secure 1000 is protected only by its modular, isolated design and by the simplicity of its firmware.

5.4.3 Privacy Side-Channel Attack

AIT screening raises significant privacy concerns because it creates a naked image of the subject. Scans can reveal sensitive information, including anatomical size and shape of body parts, location and quantity of fat, existence of medical conditions, and presence of medical devices such as ostomy pouches, implants, or prosthetics. As figures throughout the chapter show, the resulting images are quite revealing.

Recognizing this issue, the TSA and scanner manufacturers have taken steps to limit access to raw scanned images. Rapiscan and DHS claim that the TSA machines had no capacity to save or store the images [92, 131]. The TSA also stated that the backscatter machines they used had a “privacy algorithm applied to blur the image” [140]. We are unable to verify these claims due to software differences between our machine and TSA models. Our Secure 1000 has documented save, recall (view saved images), and print features and does not appear to have a mechanism to disable them. In fact, using forensic analysis software on the user console’s drive, we were able to recover a number of stored images from test scans that were incompletely deleted during manufacturing.

These software-based defenses aim to safeguard privacy in images that are constructed by the machine, but they do not address a second class of privacy attacks against AITs: an outsider observer could try to reconstruct scanned images by using their own external detector hardware. The most mechanically complex, dangerous, and energy intensive aspects of backscatter imaging are related to X-ray illumination; sensing the backscat-
Figure 5.8. Attacking Privacy — An attacker could use a detector hidden in a suitcase to capture images of the subject during scanning. As a proof of concept, we used a small external PMT to capture images that are consistent with the scanner’s output. A larger detector would produce more detailed images.

Backscattered radiation is comparatively simple. Since X-rays scatter off the subject in a broad arc, they create a kind of physical side channel that potentially leaks a naked image of the subject to any nearby attacker. To the best of our knowledge, we are the first to propose such an attack; the privacy threat model for AITs appears to have been focused almost entirely on concerns about the behavior of screening personnel, rather than the general public.

In the scenario we envision, an attacker follows a target subject (for instance, a celebrity or politician) to a screening checkpoint while carrying an X-ray detector hidden in a suitcase. As the victim is scanned, the hardware records the backscattered X-rays for later reconstruction.
We experimented with the Secure 1000 to develop a proof-of-concept of such an attack. The major technical challenge is gathering enough radiation to have an acceptable signal/noise ratio. The Secure 1000 uses eight large photomultiplier tubes (PMTs) — four on either side of the X-ray generator — in order to capture as much signal as possible. For best results, an attacker should likewise maximize observing PMT surface area, and minimize distance from the subject, as radiation intensity falls off quadratically with distance. To avoid arousing suspicion, an attacker may be limited to only one PMT, and may also be restricted in placement.

To determine whether external image reconstruction is feasible, we used a small PMT, a 75 mm Canberra model BIF2996-2 operated at 900 V, with a 10 cm × 10 cm NaI crystal scintillator. We placed this detector adjacent to the scanner and fed the signal to a Canberra Model 1510 amplifier connected to a Tektronix DPO 3014 oscilloscope. After capturing the resulting signal, we converted the time varying intensity to an image and applied manual enhancements to adjust levels and remove noise.

Figure 5.8 shows the results from the scanner and from our corresponding reconstruction. While our proof-of-concept results are significantly less detailed than the scanner’s output, they suggest that a determined attacker, equipped with a suitcase-sized PMT, might achieve satisfactory quality. A further concern is that changes in future backscatter imaging devices might make this attack even more practical. Since the PMTs in the Secure 1000 are close to the maximum size that can fit in the available space, further improvements to the scanner’s performance — i.e., better resolution or reduced time per scan — would likely require increased X-ray output. This would also increase the amount of information leaked to an external detector.
5.5 Discussion and Lessons

The Secure 1000 appears to perform largely as advertised in the non-adversarial setting. It readily detected a variety of naively concealed contraband materials. Our preliminary measurements of the radiation exposure delivered during normal scanning (Section 5.7) seem consistent with public statements by the manufacturer, TSA, and the FDA [144, 28, 76, 117]. Moreover, it seems clear that the manufacturer took significant care to ensure that predictable equipment malfunctions would not result in unsafe radiation doses; in order for this to happen a number of independent failures would be required, including failures of safety interlocks specifically designed to prevent unsafe conditions.

However, the Secure 1000 performs less well against clever and adaptive adversaries, who can use a number of techniques to bypass its detection capabilities and to attempt to subvert it by cyberphysical means. In this section, we use the device’s strengths and weaknesses to draw lessons that may help improve the security of other AITs and cyberphysical security systems more generally.

The effectiveness of the device is constrained by facts of X-ray physics …

As discussed in Section 5.2.1, Compton scattering is the physical phenomenon which enables backscatter imaging. As the tight beam of X-rays shines upon the scene, it interacts with the scene material. The intensity and energy spectrum of the backscattered radiation is a function of both the X-ray spectrum emitted by the imaging device and the atomic composition of the material in the scene.

The Secure 1000 emits a single constant X-ray spectrum, with a maximum energy of 50 keV, and detects the intensity of backscatter to produce its image. Any two materials, no matter their actual atomic composition, that backscatter the same
approximate intensity of X-rays will appear the same under this technology. This
physical process enables our results in Section 5.3.3. This issue extends beyond the
Secure 1000: any backscatter imaging device based upon single-spectrum X-ray emission
and detection will be vulnerable to such attacks.

By contrast, baggage screening devices (such as the recently studied Rapis-
can 522B; see [116]) usually use transmissive, rather than backscatter, X-ray imaging.
These devices also often apply dual-energy X-ray techniques that combine information
from low-energy and high-energy scans into a single image. To avoid detection by such
systems, contraband will need to resemble benign material under two spectra, a much
harder proposition.

…but physics is irrelevant in the presence of software compromise.

In the Secure 1000, as in other cyberphysical screening systems, the image of
the object scanned is processed by software. If that software has been tampered with,
it can modify the actual scan in arbitrary ways, faking or concealing threats. Indeed,
the ability of device software to detect threats and bring them to the attention of the
operator is presumed in the “Automated Target Recognition” software used in current
TSA millimeter-wave scanners [141]. Automatic suppression of threats by malicious
software is simply the (easier to implement) dual of automatic threat detection. As we
show in Section 5.4.1, malware can be stealthy, activating only when it observes a “secret
knock.”

Software security, including firmware updates, networked access, and chain-of-
custody for any physical media, must be considered in any cyberphysical scanning system.
Even so, no publicly known study commissioned by TSA considers software security.
Procedures are critical, but procedural best practices are more easily lost than those embedded in software.

As early as 1991, Sandia National Labs recommended the use of side scans to find some contraband:

A metallic object on the side of a person would blend in with the background and be unobserved. However, a side scan would provide an image of the object. There are other means of addressing this which IRT is considering presently [85, page 14].

Yet TSA procedures appear to call for only front and back scans, and the device manual characterizes side scans as an unusual practice:

The Secure 1000 can conduct scans in four subject positions, front, rear, left side and right side. Most users only conduct front and rear scans in routine operations and reserve the side scans for special circumstances [115, page 3-7].

Omitting side scans makes it possible to conceal firearms, as we discuss in Section 5.3.1.

Since side scans are necessary for good security, the device’s design should encourage their use by default. Yet, if anything, the scanner user interface nudges operators away from performing side scans. It allows the display of only two images at a time, making it poorly suited to taking four scans of a subject. A better design would either scan from all sides automatically (the Secure 1000 is already sold in a configuration that scans from two sides without the subject’s turning around) or encourage/require a four-angle scan.

Adversarial thinking, as usual, is crucial for security.

The Sandia report concludes that both C-4 and Detasheet plastic explosives are detected by the Secure 1000. Attached to their report is an image from one C-4 test (Figure 5.9), wherein a 0.95 cm thick C-4 block is noticeable only by edge effects—
Figure 5.9. Naïve Evaluation — In an evaluation by Sandia National Labs, a Secure 1000 prototype successfully detects blocks of C-4 plastic explosive and Lucite attached to the subject’s chest. Observe that the detection is based almost entirely on the X-ray shadow surrounding each rectangular block, which can be reduced or eliminated by an adaptive adversary through clever shaping and positioning of contraband. Reproduced from [85].

it is outlined by its own shadow, while the intensity within the block almost exactly matches the surrounding flesh. This suggests a failure to think adversarially: since plastic explosives are, by design, moldable putty, the attacker can simply gradually thin and taper the edges of the mass, drastically reducing edge effects and rendering it much less noticeable under X-ray backscatter imaging. We describe precisely such an attack in Section 5.3.3.

The basic problem appears to be that the system, while well engineered, appears not to have been designed, documented, or deployed with adaptive attack in mind. For instance, attaching contraband to the side of the body as described in Section 5.3.1 is a straightforward attack that is enabled by scanning only straight-on rather than from all angles. However, the operator’s manual shows only example images where the contraband is clearly at the front or the back.
The other attacks we describe in Sections 5.3 and 5.4, which allow us to circumvent or weaken the advertised efficacy, privacy, and security claims, again show that the system’s designers failed to think adversarially.

**Simplicity and modular design are also crucial for security.**

The system control board implements simple, well-defined functionality and communicates with the operator console by means of a simple protocol. We were unable to compromise the control board by abusing the communication protocol. This is in contrast to the scanner console, whose software runs on a general-purpose COTS operating system.

Simplicity and modular design prevented worse attacks, but do other AITs reflect these design principles? Modern embedded systems tend towards greater integration, increased software control, and remote network capabilities, which are anathema to security.

Components should be designed with separation of concerns in mind: each component should be responsible for controlling one aspect of the machine’s operation. Communication between components should be constrained to narrow data interfaces. The Secure 1000 gets these principles right in many respects. For example, the PC software does not have the ability to command the X-ray tube to a particular height. Instead, it can only command the tube to return to its start position or to take a scan.

Our main suggestion for improving the Secure 1000’s cyberphysical security is to remove the ability for the control board firmware to override the safety interlocks (something currently needed only briefly, at scan initialization). As long as this bypass functionality is in place, the interlocks can serve as safety mechanisms but not as a defense against software- or firmware-based attacks.
Keeping details of the machine’s behavior secret didn’t help . . .

Published reports about the Secure 1000 have been heavily redacted, omitting even basic details about the machine’s operation. This did not stop members of the public from speculating about ways to circumvent the machine, using only open-source information. In an incident widely reported in the press, Jonathan Corbett suggested that firearms hanging off the body might be invisible against the dark background [32], an attack we confirm and refine in Section 5.3.1. Two physicists, Leon Kaufman and Joseph Carlson, reverse engineered the Secure 1000’s characteristics from published scans and concluded that “[i]t is very likely that a large (15–20 cm in diameter), irregularly-shaped, [one] cm-thick pancake [of plastic explosive] with beveled edges, taped to the abdomen, would be invisible to this technology” [82], an attack we confirm and refine in Section 5.3.3. Keeping basic information about the device secret made an informed public debate about its use at airports more difficult, but did not prevent dangerous attacks from being devised.

. . . but keeping attackers from testing attacks on the machine might.

To a degree that surprised us, our attacks benefited from testing on the device itself. Our first attempts at implementing a new attack strategy were often visible to the scanner, and reliable concealment was made possible only by iteration and refinement. It goes without saying that software-replacement attacks on the console are practical only if one has a machine to reverse engineer. As a result, we conclude that, in the case of the Secure 1000, keeping the machine out of the hands of would-be attackers may well be an effective strategy for preventing reliable exploitation, even if the details of the machine’s operation were disclosed.
The effectiveness of such a strategy depends critically on the difficulty of obtaining access to the machine. In addition to the device we purchased, at least one other Secure 1000 was available for sale on eBay for months after we obtained ours. We do not know whether it sold, or to whom. Also, front-line security personnel will always have some level of access to the device at each deployment installation (including at non-TSA facilities) as they are responsible for its continued operation. Given these facts, imposing stricter purchase controls on backscatter X-ray machines than those currently enacted may not be enough to keep determined adversaries from accessing, studying, and experimenting with them.

5.6 Related work

Cyberphysical devices must be evaluated not only for their safety but also for their security in the presence of an adversary [77]. This consideration is especially important for AITs, which are deployed to security checkpoints. Unfortunately, AIT manufacturers and TSA have not, to date, allowed an unfettered independent assessment of AITs. Security evaluators retained by a manufacturer or its customers may not have an incentive to find problems [103]. In the case of a backscatter X-ray AIT specifically, an evaluation team may be skilled in physics but lack the expertise to identify software vulnerabilities, or vice versa.

Ours is the first study to consider computer security aspects of an AIT’s design and operation, and the first truly independent assessment of an AIT’s security, privacy, and efficacy implications informed by experimentation with an AIT device.

Efficacy and procedures

In 1991, soon after its initial development, the Secure 1000 was evaluated by Sandia National Laboratories on behalf of IRT Corp., the company then working to
commercialize the device. The Sandia report [85] assessed the device’s effectiveness in screening for firearms, explosives, nuclear materials, and drugs. The Sandia evaluators do not appear to have considered adaptive strategies for positioning and shaping contraband, nor did they consider attacks on the device’s software. Nevertheless, they observed that side scans were sometimes necessary to detect firearms.

More recently, the Department of Homeland Security’s Office of Inspector General released a report reviewing TSA’s use of the Secure 1000 [37]. This report proposed improvements in TSA procedures surrounding the machines but again did not consider adversarial conditions or software vulnerabilities.

Working only from published descriptions of the device, researchers have hypothesized that firearms can be concealed hanging off the body [32] and that plastic explosives can be caked on the body [82]. We confirm these attacks are possible in Section 5.3 and refine them through access to the device for testing.

**Health concerns**

The ionizing radiation used by the Secure 1000 poses at least potential health risks. Studies performed on behalf of TSA by the Food and Drug Administration’s Center for Devices and Radiological Health [28] and by the Johns Hopkins University Applied Physics Laboratory [76] attempted to quantify the overall radiation dose delivered by the device. Both studies saw public release only in heavily redacted form, going so far as to redact even the effective current of the X-ray tube.

In 2010, Professors at the University of California, San Francisco wrote an open letter to John P. Holdren, the Assistant to the President for Science and Technology, expressing their concern about potential health effects from the use of backscatter X-ray scanners at airports [124]. The letter writers drew on their radiological expertise, but did not have access to a Secure 1000 to study. The FDA published a response disputing the
technical claims in the UCSF letter [95], as did the inventor of the Secure 1000, Steven W. Smith [128]. Under dispute was not just the total radiation dose but its distribution through the skin and body. In independent work concurrent with ours, a task group of the American Association of Physicists in Medicine [6] explicitly considered skin dose. The task group’s measurements are within an order of magnitude of our own, see Section 5.7.

5.7 Radiation Dose Assessment

The Secure 1000 generates low-energy X-rays (50 kVp at 5 mA tube accelerating potential) to construct its images. Although this output is low, the machine still produces ionizing radiation, and careful assessment is necessary to ensure public safety.

The imparted dose has been scrutinized recently by various agencies applying a number of experimental designs [6, 143, 62]. These findings have been consistent with manufacturer claims [117] that per-scan radiation exposure to subjects is nonzero, but is near natural background levels. Additionally, there have been claims and counter-claims surrounding the distribution of dose within the body, with some groups raising concerns that the scanner might impart a minimal deep dose but an overly large skin dose to the subject [124, 128, 28].

To shed light on this question, we executed a brief assessment of the radiological output of the scanner using Landauer Inc.’s InLight whole body dosimeters. These dosimeters give a shallow dose equivalent (SDE), a deep dose equivalent (DDE), and an eye lens dose equivalent. They are analyzed using optically stimulated luminescence (OSL), an established dosimeter technology [35, 79]. We read the results using Landauer’s proprietary MicroStar dosimeter reader.

We used a simple experimental design to quantify the dose output: we arranged 21 dosimeters on a RANDO chest phantom positioned upright on a wooden table with a neck-to-floor distance of 144 cm and a source-to-detector distance of 66 cm, approximating
the conditions of a normal scan. The dosimeters give a more accurate dose representation if the incident beam is perpendicular to the detector material. In this case, the dosimeters were attached to the chest phantom without regard for beam angle, and so no correction factors were implemented; geometry issues were expected in the results.

The InLight dosimeters require a total dose of at least 50 µSv to be accurate. To irradiate them sufficiently, we performed 4033 consecutive single scans in the machine’s normal operating mode. (Each screening consists of at least two such scans: one front and one rear.) A scan was automatically triggered every 12 s and lasted 5.7 s, for a total beam-on time of 6 h 23 min.

We read the dosimeters the following day. A small loss of dose due to fade is expected, but for the purpose of this study we regard this decrease as negligible. We applied the standard low-dose Cs-137 calibration suggested by Landauer. Initially, we were concerned that the low energy output of the scanner (50 kVp tube potential emits an X-ray spectrum centered roughly in 16 keV–25 keV) would lead to inaccurate readings on the InLights, but since the dosimeters are equipped with filters, the dose equation algorithm in the MicroStar reader can deduce beam energy without a correction factor applied to the 662 keV energy from the original calibration.

The average DDE per scan for all the dosimeters was calculated to be 73.8 nSv. The average SDE per scan was 70.6 nSv, and the average eye-lens dose per scan was 77.9 nSv. The standard deviation (σ) and the coefficient of variation (CV) value of all the dosimeters for the DDE were 0.75 and 0.10 (generally low variance) respectively. For the SDE and lens dose, σ and the CV were 1.26 and 0.16, and 2.08 and 0.29, respectively.

An unexpected aspect of our results is that the measured DDE is higher than the SDE, and this occurrence is worth further examination. The irradiation geometry of the dosimeters could possibly explain this irregularity. It might be productive to conduct further experiments that account for this effect.
The doses we measured are several times higher than those found in the recent AAPM Task Group 217 report [6], but they still equate to only nominal exposure: approximately equal to 24 minutes of natural background radiation and below the recommendation of 250 nSv per screening established by the applicable ANSI/HPS standard [7]. A person would have to undergo approximately 3200 scans per year to exceed the standard’s annual exposure limit of 250 µSv/year, a circumstance unlikely even for transportation workers and very frequent fliers.

5.8 Conclusions

We obtained a Rapiscan Secure 1000 and evaluated its effectiveness for people screening. Ours was the first analysis of an AIT that is independent of the device’s manufacturer and its customers; the first to assume an adaptive adversary; and the first to consider software as well as hardware. By exploiting properties of the Secure 1000’s backscatter X-ray technology, we were able to conceal knives, firearms, plastic explosive simulants, and detonators. We further demonstrated that malicious software running on the scanner console can manipulate rendered images to conceal contraband.

Our findings suggest that the Secure 1000 is ineffective as a contraband screening solution against an adaptive adversary who has access to a device to study and to use for testing and refining attacks. The flaws we identified could be partly remediated through changes to procedures: performing side scans in addition to front and back scans, and screening subjects with magnetometers as well as backscatter scanners; but these procedural changes will lengthen screening times.

Our findings concerning the Secure 1000 considered as a cyberphysical device are more mixed. Given physical access, we were able to replace the software running on the scanner console, again allowing attackers to smuggle contraband past the device. On the other hand, we were unable to compromise the firmware on the system control board,
a fact we attribute to the separation of concerns embodied in, and to the simplicity of, the scanner design.

The root cause of many of the issues we describe seems to be failure of the system designers to think adversarially. That failure extends also to publicly available evaluations of the Secure 1000’s effectiveness. Additionally, the secrecy surrounding AITs has sharply limited the ability of policymakers, experts, and the general public to assess the government’s safety and security claims.

Despite the flaws we identified, we are not able to categorically reject TSA’s claim that AITs represent the best available tradeoff for airport passenger screening. Hardened cockpit doors may mitigate the hijacking threat from firearms and knives; what is clearly needed, with or without AITs, is a robust means for detecting explosives. The millimeter-wave scanners currently deployed to airports will likely behave differently from the backscatter scanner we studied. We recommend that those scanners, as well as any future AITs — whether of the millimeter-wave or backscatter [112] variety — be subjected to independent, adversarial testing, and that this testing specifically consider software security.
Acknowledgments

We are grateful to the many people who contributed to the preparation of this report, including Michael Bailey, Nate Cardozo, Cindy Cohn, Prabal Dutta, Jennifer Granick, Nadia Heninger, Daniel Johnson, Daniel Kane, Brian Kantor, Falko Kuester, Sarah Meiklejohn, Kurt Opsahl, Stefan Savage, Daniel Scanderberg, Alex Snoeren, Geoff Voelker, Kai Wang, and the anonymous reviewers. We also thank the outstanding staff members at UCSD and the University of Michigan who helped us navigate the legal and regulatory landscape and made studying the device possible. This material is based in part upon work supported by the National Science Foundation Graduate Research Fellowship Program.

Chapter 5, in part, is a reprint of the material as it appears in USENIX Security 2014. Mowery, Keaton; Wustrow, Eric; Wypych, Tom; Singleton, Corey; Comfort, Chris; Rescorla, Eric; Checkoway, Stephen; Halderman, J. Alex; Shacham, Hovav, 2014. The dissertation author was a primary investigator and the primary author of this paper.
Bibliography


[21] Debra Bowen, David Wagner, Aaron Burstein, Arel Cordero, Ariel Feldman, Candice Hoke, Chris Karlof, Dan Wallach, Dave Kettyle, Davide Balzarotti,

[22] Ernie Brickell. Recent advances and existing research questions in platform security. Invited talk at Crypto 2012, August 2012.


[79] Paul A Jursinic and Clifford J Yahnke. In vivo dosimetry with optically stimulated luminescent dosimeters, OSLDs, compared to diodes; the effects of buildup cap thickness and fabrication material. Medical Physics, 38(10):5432, 2011.


