University of Calgary

PRISM: University of Calgary's Digital Repository

Science | Science Research & Publications

2012-06-13T22:23:58Z

Babel: A Secure Computer is a Polyglot

Aycock, John; Castro, Daniel Medeiros Nunes de; Locasto, Michael; Jarabek, Chris

http://hdl.handle.net/1880/49000
technical report

Downloaded from PRISM: https://prism.ucalgary.ca

Babel: A Secure Computer is a Polyglot

John Aycock, Daniel Medeiros Nunes de Castro,
Michael E. Locasto, and Chris Jarabek
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, AB, Canada T2N 1N4
{aycock,dmncastr,locasto,cjjarabe}@ucalgary.ca

Working paper: TR 2012-1026-09, June 2012

Abstract

Why should a user's computer be trusted at all? We propose a new model of the computer, Babel, that makes a user's computer appear as it normally would, but is actually untrusted to the point where it cannot run the code installed on it. Each computer, each process, speaks a different language, and a translator on the network is needed to allow a user's computer to execute code. This has enormous implications. The user gets continuous protection, and multiple kinds of protection, with no need for security updates or patches. At the same time, the user effectively has an adjustable control that they can set based on their risk assessment and need for privacy. Babel can work perfectly well alongside existing systems, and opens new markets for security.

1 Introduction

A deeply-ingrained assumption about security is that, somehow, the very computer we are trying to defend is magically reliable enough to determine whether or not it has been compromised. This is akin to asking a person whether or not they're sane – if they're not, how would they know? In computer terms, malware like rootkits provides similar uncertainty. Even if we ignore the possibility of malware actively hiding, it may coexist with security software that is blind to its existence, like anti-virus software that lags in detection or has not been updated in recent memory.

Part of the problem is that our computers have too much autonomy to execute code, legitimate or malicious, with no questions asked. Computers can get themselves and their users into a lot of trouble as a result: running malware, becoming a node in a botnet, allowing sensitive information to be sent out, letting code exploits in.

We propose instead that the user's computer be perpetually, inherently untrusted. A user's computer (desktop, mobile) looks exactly as it does now. A user can install software on their machine, and their data can be anyplace – local, remote – as it is now.

The fundamental difference is that the computer is *physically incapable* of directly executing any instructions in the installed software, whether the software is malicious or legitimate. The software must be translated piecewise by (one or more) network servers. Because each computer and even each process within the computer effectively speaks a different "language" of instructions, we refer to this system as *Babel*.

Take an extreme example. For each instruction executed, the user's computer uploads that instruction (which it can't make sense of) to a server, which translates it into a small snippet of code the computer *can* run; this process keeps iterating to execute the program. In a straightforward implementation, the translation may equate to decryption; each computer, or each process within the computer, can have its own unique decryption key known by the server, automatically adding diversity. (The key could potentially change during execution too.)

This case is but the tip of the iceberg in terms of Babel. Section 2 presents the full model along with the positive effects in terms of security. It is followed by ways to address the inevitable performance issues in Section 3. Section 4 analyzes the dark side of Babel and what (new) attacks and concerns arise, and our preliminary implementation results are given in Section 5. Finally, Sections 6 and 7 contain related work, and our conclusions and future work, respectively.

2 Babel and Its Security Effects

We can divide Babel's full model into an examination of what happens at each end (the client and server sides) and also how clients and servers interact. We follow that with some example deployment scenarios.

2.1 Client Side

The client side refers to the user's machine or device. Pragmatically, the client side must have the ability to run at least some minimal amount of software to provide a basic level of operating system support. We envision that the client side would have a very small, non-updateable kernel burned into ROM, able to provide system services and communicate on the network. In operating system terms, this would be a nano- or picokernel, not only small but possible to thoroughly audit for security flaws, or even formally verify (e.g., seL4 [36]). Its singular function with respect to code execution is to send instructions from programs, instructions it cannot interpret, to the server and execute the translated code that comes back.

2.2 Server Side

Most of the interesting work in Babel happens on the server side. The most basic server implementation would simply be a translation service, where a client sends instructions for the server to translate, decode, or decrypt into a form the client can understand. (This translation, and all server-side functionality, can be sandboxed or otherwise access-restricted to limit the effect of any software flaws in the server.)

However, translation servers have access to a stream of instructions before they are executed by the client, leaving the server uniquely positioned for many possibilities beyond rote translation. The server can act as a behavior monitor or behavior blocker, comparing the instructions in the stream to dynamic malware signatures. As a simple example, a server could watch for system calls redolent of an appending virus. Similarly, assuming the server knows what code is being executed and stores a record of its past executions, the server may perform anomaly detection [31] to flag any variance that might be attributable to malware. Depending on the amount of code that the client sends to the server at a time (Section 3), the server may also be able to check for static malware signatures [5, 52].

There is no reason the server cannot dynamically modify the code as it translates. Servers could add in code, that is run on the client side, to check and enforce data access restrictions on the client; the server can be a remote reference monitor [1], in a sense, that emits code to the client to implement the security policy *du jour*. High-security servers may automatically introduce timing noise into the translated code to make some covert channels difficult to exploit [37]. Software patches, long a bugbear, for which large companies require infrastructure to satisfy massive demand [25], may be dynamically applied instantly to all clients by pushing the updates to translation servers.

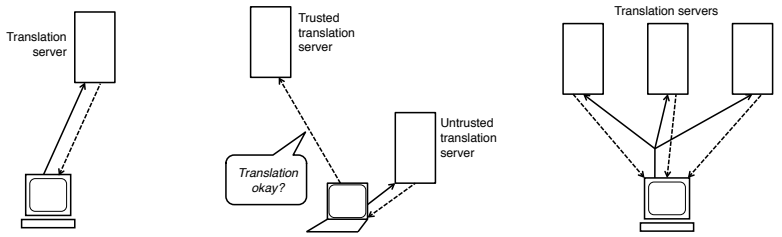
A translation server also has the advantage of scale. A single server may be translating instructions for a large number of clients, and as a result has a far more global view of what code is executing where. Perhaps equally important is the ability to detect when previously-unknown code is running; it may be indicative of a new legitimate software release, but may also flag new (or repacked) malware [42]. Furthermore, different translation servers can share information amongst themselves to construct an even bigger picture.

We have assumed thus far that the user experience from the client side would be unchanged, albeit slower. The user could install software locally, could have data both local and remote. If we relax that assumption, client-side software could reside on the server side, where the server could pre-translate it or maintain it in some form amenable to easy translation (similar in concept to slim binaries [23]), rather than perform so much work on demand. Obviously, the translation server would also be in a position to enforce software licensing agreements, a foreshadowing of problems we return to in Section 4.

2.3 Betwixt Client and Server

The Babel model offers a wide range of options when it comes to interactions between clients and servers. Foremost is the fact that different servers may have different properties:

Proximity. Although we have not discussed performance as yet, network latency is clearly going to be a problem, if not *the* problem. It stands to reason that servers that are physically located closer to the client will likely perform better for instruction translation. As it happens, suitable facilities are either in existence or being constructed: consider the distributed data centers that Google builds,



(a) Home use (b) Café use (c) High-security use

Figure 1: Sample Babel deployment scenarios. Solid lines are untranslated instructions and dashed lines are translated instructions.

for example [28]. This is also an opportunity for ISPs, particularly "last mile" providers, to offer another (monetizable) service to their customers.[1]

Security. Translation servers can have radically different security properties, from straightforward translation to high-security data access verification and addition of anti-covert-channel jitter. Some servers may have more extensive or frequently-updated malware signatures than others; some may err on the conservative side and flag more suspect bits of code at the risk of more false positives. Basically, Babel allows users choice, the ability to select a server based on their perceived risk.

Pricing. We advocate the use of an open protocol for Babel, admitting a variety of translation servers to peacefully coexist. Free, freely-accessible servers are possible, although the computing infrastructure to support a large number of geographically-disparate clients is financially daunting. Companies and organizations, of course, can run their own in-house translation servers. We also foresee servers that are subsidized by advertisement, where the server's dynamic translation would periodically insert code to display ads. A translation server (with or without ads) would fit well into a subscription model, presumably where servers with more desirable properties would command a higher price. Babel enables entirely new security markets.

Ideally, one property a server should have is trustworthiness. We routinely delegate trust without hesitation; who has not shrugged and accepted a nagging software update? In Babel, however, there is no *a priori* reason to assume that the client should trust the server to provide a correct, unadulterated translation.

A client requiring higher security can multicast their translation request to multiple servers, ensuring that they receive the same translation back from all servers; naturally, this assumes that the servers in question translate the same way. (A client wanting security and reliability may contact servers known to translate differently, for an approximation to N-version programming [3]; multivariant execution [51] is also related.) An alternative to multicast is a trust-but-verify variant. Here, a client would primarily rely

[1]Translation may be pushed out further by an ISP to cable modems or router/firewall devices the ISP supplies to customers, as they are often fully-capable computers themselves.

on a single server, but relay the resulting translation to another, more trusted server for verification.

It may, however, be unacceptable for a single server to see the full code stream from a client, in which case a client may alternate its requests between multiple servers, or send different code slices to different servers.

Again, this time through the possibility for using multiple translation servers, Babel allows security to be adjusted appropriately to risk.

2.4 Example Scenarios

To make Babel's model more concrete, we consider three possible (but by no means exhaustive) deployment scenarios.

First is the case of a typical home user (Figure 1a). It is not a high-security environment but a low-maintenance one, where the user may have a lack of skill, inclination, or time to maintain security on their computer, so they delegate that task to Babel's translation server. A single server, perhaps part of an ISP service, is sufficient here from a trust and performance point of view.

Figure 1b illustrates the second scenario, a café and a mobile device. The overall need for security may be the same as for the home user, but the context is riskier: the nearest, best-performing translation server is untrusted and belongs to the café. The client, noting the change in wireless network, switches to a trust-but-verify configuration; the café server is used for translation and running code, but a copy of the café server's translations is transmitted (by the client's kernel) to an outside, trusted, but more distant server that verifies the untrusted translation. The outside server cannot block code execution in this case, but can signal the client that a malicious/suspect translation has been given to it.

Finally, the third case (Figure 1c) is a high-security deployment, such as a bank, a government, or a military installation. The client multicasts to multiple translation servers, which perform the same translation but are under different administrative control; this yields not defense in depth, but defense in breadth. The higher-security translations dynamically add access control checks according to the site security policy, and some jitter to confound covert channel use.

3 Making Babel Faster

The obvious performance bottleneck for Babel is network latency. On one hand, this can be dismissed: Babel targets the networks of the future more so than the present. On the other hand, there are a number of optimizations that can be employed to bring this "future" a little closer. We can categorize these optimizations in three groups.

First, the round-trip time between client and server can be reduced. It is reasonable to expect that the client would maintain a local translation cache that would store translations made by the server for hot traces like loops (like the fragment cache in [8]); this reduces those round-trip times to zero. The server could also maintain a translation cache if necessary, although dynamic translation time is likely to be dwarfed by

network latency. An intermediary cache, similar to a web cache [7], may also be useful for translations of commonly-executed code.

Second, we can translate larger chunks of code at a time than single instructions. This can be a boon to performance, as our WAN results in Section 5 show. This can leverage groupings of instructions used in compiler technology – basic blocks, extended basic blocks [41], slices [56, 58], even functions – but it is not yet clear which is the best choice. There is an additional problem, that the client would have to have some understanding of the code in order to (for example) break code into basic blocks precisely, unless the code is appropriately prepackaged for the client, or the client can use some heuristic instead, like always sending the next N bytes of instructions. A related optimization is branch prediction [38, 39], where the server might return extra translated instructions in anticipation of a control flow change.

Third, we can reduce the amount of network traffic. There is prior work on small instruction sets that are compressed [11] or specialized to particular applications [22], which accomplishes the goal of reducing network traffic but also fits in well with Babel's theme of multiple instruction set "languages." Another option is where the server, upon being asked to translate a chunk of instructions, either dynamically optimizes the code to make it smaller [4] or (if there are no side effects) executes the code on the server side and simply sends the result back to the client. The latter would be attractive particularly for resource-constrained mobile devices.

4 Attacks and Concerns

As might be expected, Babel is no silver bullet and may be attacked in a number of ways. Looking first at traditional attacks, the Achilles' Heel of Babel is obviously its reliance on network access. Any successful denial of service affecting the network would not compromise security *per se*, but would render Babel unusable. Similarly, an active eavesdropper able to garble Babel traffic would yield a similar but more targeted result.

A man-in-the-middle attack is another traditional concern. A client would definitely want to authenticate the server. Would a server need to authenticate clients? Sometimes. A for-profit translation server would to avoid freeloaders, and a high-security server would to make it more difficult for an adversary to see what additional features are being added to translated code.[2] Depending on the perceived level of risk, a verification server that can be queried by an adversary leaves open the possibility that an adversary will learn how they can circumvent the translation verification, akin to malware writers testing against anti-malware software or spammers testing against anti-spam software. Here again, the verification server would want to authenticate the client. However, a free, open-source translation server, or an organization's internal-only server could conceivably have no such need.

Passive eavesdropping is again traditional, a threat that can be countered by use of encryption for all Babel communications. From an OPSEC point of view, the mere fact that a device is in use – even if the details cannot be discerned – gives away information

[2] We emphasize "more difficult" rather than "impossible" as this is really just security through obscurity. But perhaps that's not a such a bad thing: see Pavlovic [46].

to an adversary. Babel is especially vulnerable in this respect, because even computer usage that would currently be "local only" would need instruction translation and thus network traffic in Babel. This is another area where Babel can be adjusted based on risk assessment: a typical home user would likely have no need to care about this weakness, but a high-security deployment could inject chaff into the Babel communications. A related issue is that instruction execution's external manifestation on the network may provide another form of covert channel, and a high-security version may require countermeasures, like those discussed in [24].

Nothing is more traditional than exploitable bugs, of course. The programs being translated have no guarantee that they are devoid of bugs. The advantage Babel brings to the table in this regard is – as mentioned – the ability to dynamically patch a flawed program as it executes.

Security at the server endpoint is paramount, but perhaps the bigger concern is one of privacy. We are now accustomed and maybe even inured to our data being collected and mined for advertising or other motives; our activity on keyboard, mouse, webcam, and microphone being a target for yet more data gathering [6]; even our inactivity possibly being telling [30]. Babel allows servers the potential to go a step further, to be able to construct a complete picture of everything happening on a client computer down to the instruction level. High-security deployments may view this extra ability to monitor activity as boon and not bane, but this opinion may not be widespread outside this niche.

Countermeasures that can be taken on the client side would likely take one of two forms, trading performance for privacy. First, as mentioned in Section 2.3, a client could try to avoid giving a single server a complete view of code execution by switching between servers or sending only partial code slices (but servers could still collude). Secure program partitioning [61] may find application here as well. A programming style for applications involving multiple threads of execution with a high degree of inter-thread communication [15] may make the job of a nosey server more difficult too. Second, a client may try to hide its identity using an anonymity network (e.g., Tor [18]). Recent work fingerprinting machines with surprising accuracy using leaked incidental information [20, 60] may make this a fool's game, though. Until a full Babel system is built, it is hard to ascertain with any precision how effective these defenses, or how grave the privacy breaches, will be.

One final concern is that the ability to closely monitor execution may lead to mission creep, demands to watch for more than attacks against the user and malware running on the user's computer. Enforcement of software licenses, mentioned earlier, is one distinct possibility. The potential to record all actions on a computer, accidental actions, actions started but thought better of and not taken, is troubling and warrants more consideration.

5 Preliminary Experiments

The goals of our initial proof-of-concept implementation were twofold. First, we wanted to know how programs would perform, more specifically how Babel's network activity would impact performance. Since the network activity will dominate the

Table 1: Number of instructions

	Instruction Counts	
Program	Static	Dynamic
pwd	30	62
ls	521	35364
fortune	201	2629
md5sum	117	30526

Table 2: Execution time (ms) without Babel

Program	Average	Std.Dev.	Median
pwd	0.73	0.63	1.00
ls	12.03	0.48	12.00
fortune	319.73	1.21	320.00
md5sum	48.40	8.30	53.00

results, we have used a virtual machine on the client side rather than invest time developing a custom kernel yet. Second, we wanted to use the initial implementation to "smoke out" hidden problems in the Babel model and see how they might be addressed.

For these preliminary experiments, we have implemented a basic Babel client using Dis [57, 59], the virtual machine for the Inferno [19] operating system. Dis is a memory-centric virtual machine, primarily using memory addresses for operands,[3] with fixed-size,[4] 12-byte long instructions. System calls, apart from a few operations like thread creation and termination that are explicit VM instructions, manifest themselves as functions implemented within a `Sys` module. This latter feature unfortunately complicates the job of Babel's translation server when monitoring system calls, requiring extra communication to ascertain which module and which function are being called. We begin there, at the translation server implementation, then move to the client side, the client-server interaction, then the results.

5.1 Server Implementation

Our translation server is written in Python and has two main tasks: translation of instructions and detection of malicious activities. The translation is performed on basic blocks, i.e., sequences of straight-line code without any jump or conditional branch instructions.

Three different types of detection have been implemented. First, static analysis. Since our server is currently not privy to the full executable image or large chunks of code (chunk sizes are discussed in Section 5.4), and does not have direct access to the client's memory, our prototype system instead allows the server to occasionally request data from the client. This data may be in the form of bytes of the client's

[3] As opposed to registers or a stack.

[4] Despite what the VM specification says.

memory or a hash thereof (for privacy), which the server compares to a list of static malware signatures. Second is dynamic analysis, where the server monitors sequences of system calls for malicious activity. Effectively this means the server is looking for dynamic malware signatures. Third is a hybrid static/dynamic approach, where the server looks for static signatures in the stream of instructions that are being translated. This is very fine-grained monitoring, and as we translate blocks of instructions, the server may actually detect a signature some instructions in advance of their execution on the client.

5.2 Client Implementation

On the client, Dis was modified to create a communication channel to the translation server using Inferno's Styx protocol [47] when the system starts. The client uses that to implement the communication protocol described below (Section 5.3).

Inferno normally has a virtual machine for all the processes. As Babel needs a different instruction set per process, we also modified Dis such that each process has an associated table, mapping instruction opcodes to the functions within the VM implementation that implement them, allowing us to change the mapping (Section 5.3). The original Dis bytecode is thus unrunnable by a remapped instruction set, which is where the server's translation comes to the rescue. As we are interested in the network overhead, the dominant factor, the server's translation is simply an exclusive OR currently.

5.3 Client-Server Interaction

When booting the operating system, a Babel client connects to the translation server via TCP. This connection persists during the entire execution of the operating system, and the client will attempt to reconnect if the connection is dropped.

The client and server communicate via messages, which are listed in Appendix A; in general, a message `GET`*`X`* is replied to by an *`X`* message. (Future protocol versions will use a denser, binary format for the messages too.)

The client requests a new VM for each process that is started. When our server receives a `GETVM` message, it creates a unique sequential `vmid`, randomly selects a key, stores the `vmid` and key for future translations, and sends a `VM` reply. The client is then able to construct an instruction opcode mapping table for that process.

5.4 Results

The client machine in all cases, and the server in the localhost experiments, was an Intel Celeron 1.5GHz (32 bit) with 2GB RAM, running Ubuntu 10.04, kernel 2.6.32-40 (32 bit). The server for the LAN experiments was an Intel Core i5 2.5GHz (64 bit) with 8GB RAM, running Mint 12, kernel 3.0.0.14 (32 bit with PAE). Finally, the server for the WAN experiments was an Intel Core2 2.4GHz (64 bit) with 4GB RAM, running Scientific Linux 6.1, kernel 2.6.32-131 (64 bit).

Table 3: Local Babel execution (times in ms)

Program	Chunk Size	Average	Std.Dev.	Median
pwd	60	40.47	0.92	40.00
pwd	120	40.83	1.07	41.00
pwd	240	42.47	0.72	42.00
ls	60	9445.03	63.99	9432.00
ls	120	9675.10	37.21	9674.00
ls	240	10697.80	36.59	10699.00
fortune	60	852.23	182.80	776.50
fortune	120	839.37	202.57	763.50
fortune	240	824.70	172.87	767.50
md5sum	60	7889.57	32.86	7889.00
md5sum	120	7466.37	30.05	7460.00
md5sum	240	7978.30	92.96	7958.00

Table 4: Bytes transmitted and chunk sizes

Program	Chunk size		
	60 bytes	120 bytes	240 bytes
pwd	5653	7159	10759
ls	3195243	4605456	9846191
fortune	186472	345876	457416
md5sum	2463263	3616346	4629139

In terms of networking, the LAN was wired, 100Mbps. For the WAN configuration, the client side was 25Mbps download and 2.5Mbps upload, maximum (the server side was much faster, but the client side was the limiting factor here).

The benchmark programs and their properties are given in Table 1, and Table 2 shows the time to execute each without Babel. The "ls" command was run on a directory containing 100 files, and "md5sum" was run on 100 files with 1124 bytes total. All times reported here are measured in milliseconds, and all experiments are repeated 30 times. The number of executed instructions include those within functions from external modules, such as "system calls."

The initial scenario consists of local execution, i.e., both client and server run on the same physical machine. The network latency is minimized in this case. The average time for local execution is shown in Table 3.

The latency is obviously relative to the amount of data being transmitted. We performed experiments with different chunk sizes. As the server currently replies with the instructions within a single basic block only, excess data sent is ignored by the server. So, bigger chunks would increase the amount of data without bringing any advantage to the execution.[5] The amount of bytes transmitted for each chunk size is listed in

[5]Unless larger translations can be done. Some earlier experiments we did used extended basic blocks that

Table 5: Babel WAN execution (times in ms)

Program	Chunk Size	Average	Std.Dev.	Median
pwd	60	837.30	111.05	798.50
pwd	120	828.50	51.24	822.50
pwd	240	672.63	23.81	672.00
ls	60	203884.30	16857.89	199721.50
ls	120	183539.67	5472.08	183093.00
ls	240	169911.10	2460.09	169632.50
fortune	60	14779.33	2764.76	14664.00
fortune	120	12627.30	2395.97	12199.00
fortune	240	11416.33	2106.74	10858.00
md5sum	60	161536.53	6096.49	161543.00
md5sum	120	145680.37	4704.59	145590.00
md5sum	240	124914.23	5985.44	123146.50

Table 4.

Comparing the local execution with the original execution (without Babel), we notice that the system becomes, depending on the program being executed, up to about 900 times slower. This apparently huge delay is, in fact, comparable to the difference between interpreted and compiled code execution [49]. As our system essentially performs code interpretation, seeing this delay for an initial implementation is not unacceptable.

However, the latency when working in a WAN environment (Table 5) causes our system to become almost unresponsive. An action as simple as listing a directory of 100 files takes about 3 minutes to complete. This scenario clearly points to the need for better network access.

This argument can be demonstrated by the latency in a LAN environment (Table 6), with much higher bandwidth than the WAN environment. In these experiments, we were able to achieve performance close to the local execution. In fact, in some cases LAN execution *did* perform better than the local execution. Further observation indicates that the server specification (Intel i5 2.5GHz), compared to the client (Intel Celeron 1.5GHz), favored the LAN experiments over the local, when the amount of communication was relatively small.

Some latency issues are no doubt exacerbated by the need for callbacks to allow the server to identify system calls, which is an unexpectedly nontrivial process in the Dis virtual machine. Table 7 shows the number of callbacks for each benchmark program. Solutions we are considering range from changing virtual machines, to modifying Dis, to implementing a trusted client-side monitoring thread that provides extra information without extra traffic.

---

could be guarded with conditional instructions and thus be longer. These reduced the network latency and increased the chance of a client getting a hit in the local translation cache. Unfortunately, they have been temporarily removed to facilitate detections on the server side.

Table 6: Babel LAN execution (times in ms)

Program	Chunk Size	Average	Std.Dev.	Median
pwd	60	36.63	1.62	36.00
pwd	120	37.77	0.62	38.00
pwd	240	41.20	0.83	41.00
ls	60	10803.70	292.88	10744.50
ls	120	11151.50	400.29	11038.00
ls	240	13110.87	448.89	12868.50
fortune	60	669.93	300.92	588.00
fortune	120	712.87	395.83	545.00
fortune	240	756.07	295.86	647.00
md5sum	60	9098.47	468.32	9247.50
md5sum	120	8463.97	603.99	8506.50
md5sum	240	9343.23	264.14	9396.00

Table 7: Number of callbacks

Program	Callbacks required
pwd	11
ls	128
fortune	210
md5sum	2366

6 Related Work

There is relatively little directly related work that we have found. In a way, this is not surprising, as networks are only just arriving at the point where they can start handling tasks like Babel.

Historically, there are likenesses to thin clients and dumb terminals, except Babel clients are not simply I/O devices. Parallels to the idea of a computer utility like Multics [16][6] can be seen, but while the need for a computer utility for the general public may not have been established back then, the need for computer security for the general public is firmly established now, as are security business models involving ongoing subscriptions. Also loosely related is the old NeWS windowing system [29], as that allowed programs to be implemented with code on both the server and client, and the client sent code to the server.

The closest historical reference is perhaps Thimbleby [55], who envisioned a means for standalone computers to deal with viruses, by making 'each machine unique: incompatible with everything else' [55, page 112]. His method presaged instruction set randomization, but also made patching software 'practically impossible' and was never suggested to counter any threats beyond viruses or to be used on networked machines. His model also did not prevent a computer from running a program in its entirety, once it was installed and was privy to the machine's 'password.'

On the client side, our need for a tiny kernel suggests the applicability of microkernels, some classic examples of which are Chorus [50], Mach [27], and MINIX [53]. There is other related work in the sense of building a very small kernel (e.g., exokernel [21] or JX [26]) and/or a secure kernel (e.g., Singularity [32]). Google's Chrome OS [54], a.k.a. Chromium OS, also provides a lightweight operating system on the client side, effectively making the browser the operating system. This is heavyweight in comparison to Babel's client side. Also, Chrome OS assumes that software and data are in the cloud, whereas Babel is not restricted that way.

On the server side, security software has been moving to the cloud for several years now, especially in the anti-virus area (e.g., [13, 42]). This can be manifested as a traditional anti-virus scan simply pushed into the cloud. CloudAV [43, 44], for example, has a lightweight anti-virus client on the end host whose anti-virus is a local cloud service running multiple anti-virus programs. Martignoni et al. [40] built a cloud-based behavioral analysis system that enhanced the realism of the analysis environment by shunting selected system calls back to an end user's machine, but only for detection purposes; the code was not actually being run on the client side as Babel does.

Other systems, developed for mobile devices, have looked to offload intensive security processing from smartphone to cloud. One of CloneCloud's applications is performing virus scanning in the cloud [14], but apart from this application, the overall thrust of their work is load balancing in general and not security. Jakobsson and Juels [33] suggest sending activity logs from a client to trusted servers, but their post-mortem detection lacks immediacy of protection, and even their notion of doing the log analysis in real time would be less powerful than Babel's ability to view and modify the

[6] A contemporary 1965 Multics-related paper presciently remarked: 'If every significant action is recorded in the mass memory of a community computer system … the daily activities of each individual could become open to scrutiny.' [17, page 245]. This relates to our privacy concerns in Section 4.

full instruction stream. Paranoid Android [48] replicates the state of an entire mobile device in a virtual environment on a server, by recording system calls on the mobile device and transmitting them to the server for replication. The server can then apply multiple (heavyweight) security analyses on its replica. Babel's model is much broader than this; with access to view and possibly alter the full instruction stream trace, a Babel translation server can implement Paranoid Android as a subset of its functionality. In addition, potentially sensitive local data need not leave the client side in Babel.

There are clear connections between Babel's translation process and instruction set randomization (ISR), where "instruction set" must be interpreted liberally, as the technique has been shown to be applicable for higher-level languages as well [12]. Kc et al. [34] proposes a hardware based implementation, in which a special register is used to store a encryption/decryption key. Code is decrypted during execution by a XOR operation between the instruction loaded and the aforementioned key. RISE [9, 10], on the other hand, does not require special hardware. RISE's implementation using Valgrind encrypts the executable with a pseudorandom key (of arbitrary size, generated for each process) during load time and decrypts it during runtime. A key distinction between Babel and ISR is that Babel is trying to protect computers from themselves as well as outside attackers, and the general Babel model can provide a wide variety of security checks that ISR alone cannot.

Babel may also be seen from other viewpoints. First, as a kind of program shepherding [35], except Babel's shepherding process is remotely-located and is able to watch for far more security breaches than just control flow attacks. Second, it may be categorized as software-as-a-service (SaaS), but we argue that that labeling is incorrect. Following the definition given by Armbrust et al. [2, page 50], there admittedly is some overlap in our Babel variant where software is stored on the server, but this minor variant is not the main idea of Babel, where we want the user's ability to have local software and data unimpinged. While their definition of SaaS includes local software execution, they qualify that by restricting it to situations where they 'run software locally but control use via remote software licensing.' While Babel's servers are in a position to enforce licensing, it is by no means their primary task. Third, Babel can pessimistically be seen as the ultimate in "walled gardens" [45], and while Babel could be applied in the controlling way that implies, this is not intrinsic to the Babel model, which can equally well be applied in an open manner.

7 Conclusion and Future Work

Babel is a fundamental transformation of what a user's computer is. By making the user's computer inherently untrusted, incapable of running its own code, we open the door to an untapped landscape of possibilities. Translation servers can provide continuous monitoring of code that the user runs, can exploit scale, and can alleviate the need for security updates and patches being continually thrust upon the user. Furthermore, the user need not trust a single translation server, and is empowered to select configurations that reflect their (or their organization's) risk and privacy posture. Babel plays nicely beside existing systems, and will work even if everyone isn't using the Babel model.

Admittedly, there are research challenges yet to be addressed. Performance, in particular – we will be working to mitigate this using techniques mentioned here, but ultimately the network will have to catch up for Babel to be fully deployed. Power-constrained mobile devices will also be problematic due to the necessity of constant transmission and reception. New attacks may be possible. This is only the beginning of the road for Babel.

8 Acknowledgments

The authors' research is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada via ISSNet, the Internetworked Systems Security Network. Thanks to Anil Somayaji and John Sullins for helpful discussions and ideas.

References

[1] J. P. Anderson. Computer security technology planning study: Volume II, Oct. 1972. ESD-TR-73-51, Vol. II.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[3] A. Avižienis. The *N*-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[4] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[5] J. Aycock. *Computer Viruses and Malware*, volume 22. Springer, 2006.

[6] J. Aycock. *Spyware and Adware*, volume 50. Springer, 2010.

[7] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. World Wide Web caching: The application-level view of the Internet. *IEEE Communications Magazine*, 35(6):170–178, 1997.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[9] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, Feb. 2005.

[10] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *10th*

ACM Conference on Computer and Communications Security, CCS '03, pages 281–289, 2003.

[11] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, Sept. 2003.

[12] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing*, 7(3):255–270, 2010.

[13] M. Chiriac. Tales from cloud nine. In *Virus Bulletin Conference*, pages 1–6, 2009.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *EuroSys '11*, pages 301–314, 2011.

[15] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, Department of Computer Science, 1997.

[16] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS Fall Joint Computer Conference*, pages 185–196, 1965.

[17] E. E. David, Jr. and R. M. Fano. Some thoughts about the social implications of accessible computing. In *AFIPS Fall Joint Computer Conference*, pages 243–247, 1965.

[18] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium*, 2004.

[19] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *42nd IEEE International Computer Conference*, COMPCON '97, pages 241–244, 1997.

[20] P. Eckersley. How unique is your web browser? In *10th Privacy Enhancing Technologies Symposium*, pages 1–18, 2010.

[21] D. R. Engler. *The exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, 1998.

[22] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *ACM Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

[23] M. Franz and T. Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[24] C. G. Girling. Covert channels in LAN's. *IEEE Transactions on Software Engineering*, SE-13(2):292–296, 1987.

[25] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vojnovic. Planet scale software updates. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 423–434, 2006.

[26] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *USENIX Annual Technical Conference*, pages 45–58, 2002.

[27] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Summer 1990 USENIX Conference*, pages 87–95, 1990.

[28] Google. Google data centers. http://www.google.com/about/datacenters, Last accessed 25 March 2012.

[29] J. Gosling, D. S. H. Rosenthal, and M. J. Arden. *The NeWS Book: An Introduction to the Network/extensible Window System*. Springer, 1989.

[30] Q. Guo and E. Agichtein. Ready to buy or just browsing? Detecting web searcher goals from interaction data. In *33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 130–137, 2010.

[31] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[32] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: challenges and opportunities. In *10th conference on Hot Topics in Operating Systems*, 2005.

[33] M. Jakobsson and A. Juels. Server-side detection of malware infection. In *2009 New Security Paradigms Workshop*, pages 11–22, 2009.

[34] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *10th ACM Conference on Computer and Communications Security*, pages 272–280, 2003.

[35] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, 2002.

[36] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[37] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[38] J. Lee and A. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, Jan. 1984.

[39] T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. OS-aware branch prediction: Improving microprocessor control flow prediction for operating systems. *IEEE Transactions on Computers*, 56(1):2–17, 2007.

[40] L. Martignoni, R. Paleari, and D. Bruschi. A framework for behavior-based malware analysis in the cloud. In *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*, pages 178–192, 2009.

[41] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[42] C. Nachenberg, Z. Ramzan, and V. Seshadri. Reputation: A new chapter in malware protection. In *19th Virus Bulletin International Conference*, pages 185–191, 2009.

[43] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: executable analysis in the network cloud. In *Proceedings of the 2nd USENIX workshop on Hot topics in security*, pages 5:1–5:5, 2007.

[44] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *17th USENIX Security Symposium*, pages 91–106, 2008.

[45] N. Paterson. Walled gardens: the new shape of the public Internet. In *iConference 2012*, pages 97–104, 2012.

[46] D. Pavlovic. Gaming security by obscurity. In *2011 New Security Paradigms Workshop*, pages 125–139, 2011.

[47] R. Pike and D. M. Ritchie. The styx architecture for distributed systems. *Bell Labs Technical Journal*, 4(2), 1999.

[48] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, 2010.

[49] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, 1996.

[50] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, 1988.

[51] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 8(4):588–601, 2011.

[52] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[53] A. S. Tanenbaum. *Operating systems: design and implementation*. Prentice Hall, 1987.

[54] The Chromium Project. Chromium OS. http://www.chromium.org/chromium-os, Last accessed 3 April 2012.

[55] H. Thimbleby. Can viruses ever be useful? *Computers & Security*, 10(2):111–114, 1991.

[56] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[57] Vita Nuova. Dis virtual machine specification, 9 January 2003, last accessed 5 April 2012.

[58] M. Weiser. Program slicing. In *5th International Conference on Software Engineering*, pages 439–449, 1981.

[59] P. Winterbottom and R. Pike. The design of the Inferno virtual machine, Last accessed 5 April 2012.

[60] T.-F. Yen, Y. Xie, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *19th Annual Network & Distributed System Security Symposium*, 2012.

[61] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, Aug. 2002.

A Message Types

GETVM

Format: `GETVM,pid`
Client requests a virtual machine specification. `pid` is the process id in the client machine, currently for logging only.

VM

Format: `VM,type,vmid,data`
Response for a `GETVM` message. This message currently returns `XOR` as the only valid type, as the VM is created by XOR operations. `vmid` is the unique identifier for that virtual machine, and `data` contains the key used to encrypt the VM.

GETCHUNK

Format: `GETCHUNK,data`
Sends a "chunk" of data for translation.

CHUNK

Format: `CHUNK,data`
Returns a block of translated instructions. In Inferno, each instruction is 12 bytes long. We send an extra byte containing flags, which consist of markers for special operations that the client must perform. Currently, we only use 1 bit for callback operations, as described below.

GETCB

Format: `GETCB,instruction,pc`

When a translated instruction is marked as "callback", before its execution the client should send a `GETCB` message to the server. This provides an opportunity for the server to obtain current information about the client status before a sensitive instruction is performed. `instruction` contains the instruction to be executed and `pc` is the program counter that refers to that instruction.

CB

Format: `CB,pc`
Response for `GETCB`, indicating that the callback operation is finished.

GETDATA

Format: `GETDATA,address,type_request`
This message allows the server to request extra data from the client. `address` identifies where in the client memory the data should be gathered and `type_request` identifies the format of the data. Currently, the server can request a string, a list of strings, or a hash value. This is used, for example, when the server detects that there is a call to a function within a module and needs extra information to decide if that is a system call (i.e., a call to module `Sys`). It could also be used to check for static malware signatures in the client's memory.

DATA

Format: `DATA,data`
Response for `GETDATA`, where the client sends `data` from its current state to the server.

FINISH

Format: `FINISH,vmid`
Finishes the execution of a VM within the server.

FINISHED

Format: `FINISHED,vmid`
Simply acknowledges that a `FINISH` message was received and processed.