Safe and Automated Live Malware Experimentation on Public Testbeds*

Abdulla Alwabel
USC/ISI

Hao Shi
USC/ISI

Genevieve Bartlett
USC/ISI

Jelena Mirkovic
USC/ISI

Abstract

In this paper, we advocate for *publicly accessible* live malware experimentation testbeds. We introduce new advancements for high-fidelity transparent emulation and fine-grain automatic containment that make such experimentation safe and useful to researchers, and we propose a complete, extensible live-malware experimentation framework. Our framework, aided by our new technologies, facilitates a qualitative leap from current experimentation practices. It enables specific, detailed and quantitative understanding of risk, and safe, fully automated experimentation by novice users, with maximum utility to the researcher. We present preliminary results that demonstrate effectiveness of our technologies and map the path forward for public live-malware experimentation.

1 Introduction

Today's malware is very versatile, quickly evolving and professionally developed for a large set of thriving underground markets. Malware analysis thus attracts many researchers, especially live analysis, where malware is allowed to interact with the Internet. Such analysis is inherently risky, as traffic that malware sends to the Internet may cause harm—it may create denial-of-service attack on some remote target, scan it, send spam to it, or attempt to exploit it. But such analysis is also extremely useful to researchers, as malware may communicate with a bot-master, or other bots in the same botnet, which enables analysis of such communication and botnet infiltration. Moreover, this sort of communication might serve as a trigger to malicious behavior of interest to researchers.

Tools that manage a malware's outside communication must balance risk and research utility and are still in their infancy. Presently, state-of-the-art malware experimentation is an elite undertaking. It is conducted by a few expert research groups in specialized environments developed by them, and closed to outsiders. These specialized environments often require expert human intervention to identify malware communication or set policies [16, 13, 5, 14]. Such solutions do not scale and cannot guarantee safety.

Case for public live-malware experimentation. We argue that today's malware research is held back by the manual expert effort required and the exclusivity of experimentation environments. We take the position that we can and should work towards *publicly accessible* live malware experimentation with automated mechanisms for managing risk, which scale to the sheer volume of new and sophisticated malware, and offer low-barrier to entry to experts in other domains who wish to foray into malware research. Requiring very specific expertise and specialized customization not only limits the researcher pool, but also stifles advancement of live malware experimentation and tools. By pooling together resources and offering a publicly accessible environment for malware experimentation we can gain better cross-domain efforts which have the potential to be transformative to malware research.

While live malware experimentation on public testbeds may be considered controversial, malware analysis is a growing field that attracts many novices. The community needs a safe and accessible path for these researchers to experiment with malware and build their expertise. Without such a path, novices may resort to naive and unsafe experimentation that jeopardizes their institutions and increases the risk to the Internet.

Missing pieces. We identify two missing pieces required to support public live malware experimentation. First, and most important, automated and flexible containment mechanisms are needed. Such mechanisms must have a way to provide real or seemingly-real malware communication with external world, sufficient to elicit all useful malware behaviors. These mechanisms must allow real malware communication with botmasters and other bots, and they must not require a human expert in the loop to make decisions— i.e. which malware communication should be allowed on to the Internet—these decisions must be done in automated

*This work is in part supported by the DHS grant #N66001-10-C-2018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security.

fashion to provide scalability and non-expert access. Finally, the risk of allowed malware communication must be precisely quantified, measured and managed.

We propose an automated approach to generating fine-grained containment policies, that are customized for each connectivity check, and that can automatically evolve as more observations are gathered from malware analysis. In addition to this policy generation, we also develop a smart mimicking engine that learns a profile for each public server contacted by malware and replicates the server locally, so future communication can be contained while creating an illusion of full connectivity for malware.

We further propose a risk measurement and management engine that monitors outgoing malware traffic and how this communication is consumed by the Internet. Our management engine uses these observations to quickly detect policies that lead to potential harm to external hosts and modifies these policies to increase containment level.

The second missing piece needed for public live malware experimentation is hi-fidelity emulation of physical machines. Several malware analysis tools develop by experts are released publicly, such as [9, 23, 6]. Such tools are invaluable to support malware analysis in public testbeds by novice users. But many of these tools use CPU or system emulators to facilitate fine-grained dissection of malware functionalities (e.g., Anubis [6], TEMU [23], and Bochs [17]). Such analysis also protects the host, by isolating it from potentially harmful malware actions. But there are subtle differences between operation of emulated environments and virtual machines and those of *bare-metal* physical machines. Malware routinely checks for these differences to detect that it is being run in a virtual machine, and modifies its behavior to thwart analysis.

We propose new ways to comprehensively identify cases where virtual and emulated environments execute CPU instructions differently from bare-metal machines, and how to compensate for these differences to hide them from malware. Our approach is test-driven, fine-grained and automated. It can enumerate differences between a hardware and a virtual machine effectively and efficiently. Compared to the state-of-the-art Red Pill[19], we found five times more pills running fifteen times fewer tests.

While currently manual virtual machine modifications are needed to hide detected differences from malware, this process requires minimal code writing by a human and is done only once per each virtual machine.

In the following sections we present the architecture of our framework for live malware experimentation on public testbeds, the design and implementation details of each component and some preliminary results and experiences. While our framework and its pieces are still under development, we provide in this paper sufficient preliminary results to demonstrate its effectiveness and usefulness to researchers. Over time, our solutions will increase in automation and eventually be used safely without a human expert. This will both advance the experimentation technologies and tools, and will foster wider innovation in malware defenses.

2 Experimentation Framework

Figure 1 shows our proposed architecture for live malware experimentation on a public testbed. While the layout is similar to other frameworks (see Section 3), the functionalities of the highlighted components (shown in blue in the Figure) are novel and provide a qualitative advancement.

In our framework, all malware communication with the Internet is examined and contained by a dedicated *Warden* machine. The Warden sits between the *Inmate Network*—where malware executes on testbed machines—and the outside world. Using a fine-grained firewall and policy engine (Section 2.2), the Warden chooses one of the following actions to take with malware traffic: (1) drop, (2) rewrite, (3) rate-limit, (4) forward on to the Internet and (5) redirect to *Smart Impersonators*—services which mimic public Internet servers (Section 2.2.3).

In addition to the firewall and policy engine, the Warden supports monitoring and data persistence functionalities. These functionalities are supported through continuous collection and storing of network traces to capture all communication exchanged between the framework and the Internet. The Warden also keeps *experimental history* for each experiment—information such as the testbed user, the malware studied, the experimental environment, etc.

The Inmate Network consists of a mix of machines, some of which run VM software (e.g., QEMU [7], VMWare [8], OpenVZ [1] and Xen [21]), while others are bare metal machines. Since malware limits its behavior if it detects virtualization, machines that run VM images would also run our *Hi-Fidelity Emulator* (HFE) to defeat virtualization checks.

2.1 Hi-Fidelity Emulation

The challenges for hi-fidelity emulation are (1): create a comprehensive list of differences between a VM and a bare metal machine, and (2): create "lies" to hide these differences.

We draw on Martignoni et al.'s work on Red Pill Testing [19, 18] for enumerating differences between VMs and physical machines. Their approach is to define a state of a machine (the contents of memory, values of registers and the program counter), start with the same initial state on both a VM and a physical machine, ex-

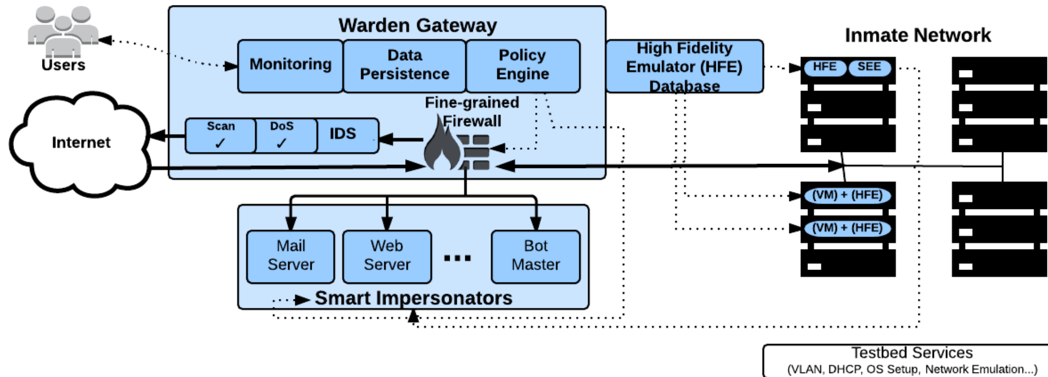

Figure 1: Framework for live malware experimentation on a public testbed.

ecute an instruction, and compare the states. A pill is found if two states differ. This process is repeated for IA-32 instructions and test cases are generated by randomly generating operands for each op code.

While these works are seminal in pill detection they have several deficiencies: (1) Random test case generation cannot guarantee that all pills will be detected. We improve on this by using instruction semantics to carefully craft test cases that explore all code paths. (2) Martignoni et al. use QEMU with Intel VT-x (hardware-assisted QEMU) as an Oracle, while we use physical machines with no virtualization. This improves fidelity of testing and ensures detection of more pills.

In our test generation, we use the semantics of each instruction to identify ranges of instruction operands that cause different execution behaviors, as reflected in different values in output registers and memory, and different exception behavior. For each range, we select both boundary and random values such that all different execution branches are examined, and all exceptions are raised.

IA-32 CPU architecture contains 906 instruction codes and a human must reason about each instruction to identify its inputs and outputs and how to populate them to test all execution behaviors. To reduce the scale of this human-centric operation we first group the instructions into five categories: arithmetic, data movement, logical, flow control and miscellaneous. Arithmetic and logical category are further subdivided into general-purpose and FPU categories based on the type of their operands. We then define parameter ranges to test per category.

Arithmetic Group. Instructions in this group first fetch arguments and then perform arithmetic operations. The arguments include actual data bits they operate on and certain flag bits that decide execution branches. We classify instructions in this group into two subgroups, depending on whether they work only on integer registers (general-purpose group), or also on floating point registers (FPU group). The instructions in FPU group include instructions with x87 FPU, MMX, SSE, and other extensions.

Based on the argument types and sizes, branch conditions, and the number of arguments, we divide both subgroups into finer partitions. For example, `aaa`, `aas`, `daa`, and `das` in the general-purpose group all compare `al` register (holding one packed BCD argument 8-bit long) with `0fh` and check the adjustment flag `af` in `efl` register. This decides the output of the instruction. To test instructions in this set we initialize `al` register to minimal (`00h`), maximal (`0ffh`), boundary (`0fh`), and random values in different ranges (`01h` $\sim$ `0eh`, `10h` $\sim$ `0feh`). We also flip `af` between clear and set for different `al` values.

Data Movement. Data movement instructions copy data between registers, main memory, and peripheral devices and usually do not modify flag bits. There are several execution branches that we explore in tests. The source and destination operands may be located outside segment limits, which may be code, data, FS, or stack segment. If the effective address is valid but paged out, a page-fault exception will be thrown. If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3, the system will raise an alignment exception. Some instructions also check direction and conditional flags, and a few others validate the format of floating point values. All these input parameters and their states that influence an instruction's execution outcome must be tested.

Logic Group. Logic instructions test relationship and properties of operands and set flag registers correspondingly. We divide these instructions into general-purpose and FPU depending on whether they use `efl` register only (general-purpose) or they use both `efl` and `mxcsr` registers (FPU). We further partition logic instructions based on the flag bits they read and argument types and sizes. When designing test cases, in addition to testing minimal, maximal, and boundary values for each parameter, for instructions that compare two parameters we also generate test cases where these parameters satisfy larger than, equal, and less than conditions.

In the FPU subgroup, we apply similar rules to generate floating point operands. We further generate test cases to populate `mxcsr` register, which has control, mask, and status flags. The control bits specify how to control underflow conditions and how to round the results of SIMD floating-point instructions. The mask bits control the generation of exceptions such as denormal operation and invalid operation. We use `ldmxcsr` to load different values into `mxcsr` and test instruction behaviors under these scenarios.

Flow Control. Flow control instructions also test condition code, alike logic instructions. Upon satisfying jump conditions, test cases start execution from another place. For short or near jumps, test cases do not need to switch the program context; but for far jumps, they must switch stacks, segments, and check privilege requirements.

Miscellaneous. Instructions in this group provide unique functionalities and we manually devise test cases for each of them that evaluate all valid and invalid use, and raise all exceptions.

We use two physical machines in our tests as Oracles: (**O1**) an Intel Xeon E3-1245 V2 3.40GHz CPU, 2 GB memory, with Windows 7 Pro x86, and (**O2**) Xeon W3520 2.6GHz, 512MB memory, with Windows XP x86 SP3. The VM host has the same hardware and guest system as the first Oracle, but it has 16 GB memory, and runs Ubuntu 12.04 x64. We test QEMU (VT-x), QEMU (TCG), and Bochs, deploying different virtualization technologies: hardware-assisted, dynamic translation, and interpretation respectively. We allocate to them the same size memory as in the Oracle. We test QEMU versions 0.14.0-rc2 (**Q1**), 1.3.1 (**Q2**), 1.6.2 (**Q3**), and 1.7.0 (**Q4**), and Bochs version 2.6.2 (**B**). The master has an Intel Core i7 CPU and installs WinDbg 6.12 to interact with the slaves. For test case compilation, we use Microsoft Assembler 10 and turn off all optimizations.

Out of 1,653 instructions present in IA-32 Intel manuals [12], with different addressing modes, there are 906 unique mnemonics. We generate total of 19,412 test cases for these instructions. Table 1 shows the breakdown of pills we found per instruction.

For each of these pills we were able to devise a method that hides its presence from malware, by modifying QEMU's translation of guest code to overwrite affected registers and memory with appropriate values that a physical machine would place there. Our future work lies in implementing these modifications in QEMU and in evaluating the completeness of our pill-finding. For comparison, we use 15 times fewer tests and discover 5 times more pills than Red Pill Testing [19]. Our testing is also more efficient, 47.6% of our test cases yield a pill, compared to only 0.6% of EmuFuzzer's tests.

2.2 Managing Malware's External Communication

Malware's external communication must be tightly managed to balance the utility of experimentation to the researcher with the risk external communication poses to the Internet. To support a range of testbed users, with differing experimental needs, traffic from each experiment needs to be subject to its own set of policies affecting external communication. But knowledge about malware behavior learned in one experiment, is shared between experiments. This allows for evolution of policies from more restrictive to permissive as testbed learns what to expect from each malware's communication.

2.2.1 Malware Management Cycle

We observe that once we allow traffic out of our framework it is impossible to guarantee that there will be no risk to the Internet. Even the most benign looking traffic, such as a single HTTP GET message, can be malicious if generated by a multitude of machines, simultaneously, to overwhelm a victim destination. We manage this risk by using the following four-step containment approach to hand each malware communication attempt:

1. Contain it and evaluate if it is a *necessary* communication for malware
2. If necessary, redirect it to a Smart Impersonator. Try Random Impersonator at first. If that does not expose sufficient malware behaviors switch to a Custom Impersonator if available.
3. If Custom Impersonator is not available, run Symbolic Execution Engine to build the Custom Impersonator.
4. If Custom Impersonator cannot be built (i.e., malware communication is *unforgeable*) let the communication out to the Internet and observe. We define different signs of this communication being harmful to the Internet in Section 2.2.4. If any of these signs is detected communication is halted.

2.2.2 Evaluation of Communication Patterns

To evaluate if a communication is necessary for malware we collect several measures of malware activity:

Category		Q1 (TCG)	Q2 (TCG)	Q2 (VT-x)	Bochs	Total tests
arith	gen	877	872	626	920	2,702
	FPU	4,525	4,486	3,603	4,245	6,743
data mov		1,788	1,780	1,524	1,804	4,394
logic	gen	371	365	346	363	2,185
	FPU	1,446	1,447	1,127	1,362	2,192
flow ctrl		164	166	169	171	1,017
misc		84	85	83	93	179

Table 1: Pills Per Instruction Category

(1) number of system calls, (2) number of unique system calls and (3) entropy of system calls. We also note if malware continues running after the communication has been denied or if it halts. If a malware sample has lower activity in isolation than when a communication is allowed or if it halts we say that this communication is necessary for live experimentation.

2.2.3 Smart Impersonators

A Smart Impersonator is a server that plays the roles of Internet hosts, such as public Web, DNS and mail servers, desktops with unpatched vulnerabilities, bot masters that communicate with our "malware inmates" interactively, etc.

A Smart Impersonator can be a Random Impersonator or a Custom Impersonator. A Random Impersonator provides valid but not specific replies to malware's communication attempts. A Random HTTP server responds with 200 OK messages with random content. A random DNS server responds to any query with its own IP address. A random SMTP server accepts messages from any user with any password. A random IRC server is a standard, out of box IRC server without any modification. Some malware may do just a communication check to see if it can reach a public server, without examining any of the content in server reply. Such malware will be tricked by a Random Impersonator to proceed with execution. Other malware may expect specific content from the server. For such communications we develop a Custom Impersonator.

This development occurs in several steps. First, we have built an Impersonation Helper that runs malware in DECAF [11], a system emulator based on QEMU [7], with a plugin we developed to collect binary traces, containing CPU instructions executed by malware and mark instructions that process data received from the network. We then translate these traces into VINE Intermediate Language[23]. We leverage VINE back-end to apply symbolic execution on the collected traces. The basic principle of symbolic execution is to replace certain values with symbols, in our case we replace network input with symbolic variables. As these values are used in computations, they produce symbolic expressions. For example, if the trace contains `sub al,bl` and al contains `0x5` and bl is marked with `t1`, then `al=5-t1`. We collect

	# TCP Flows	# UDP Flows
Number	67974	18800

Table 2: Number of Flows Generated by 390 malware samples

all symbolic expressions used in a control-flow instructions and fed them STP solver[2] to find values that satisfies or dissatisfies those expressions. These constraints tell us which checks malware performed on the network input and what changes to the network input required to execute other branches. An experimenter can then vary these parts of network input to elicit different malware behavior. Each variation that results in successful execution of malware code produces information needed to build a Custom Impersonator's profile. Such profile consists of malware requests and server replies and can be later reused by other experimenters that use the same malware binary.

Using our Impersonation Helper we have analyzed 600 samples to evaluate how frequently does malware consider communication necessary, and to investigate the trends in malware communication. Since building Smart Impersonators is currently a highly manual process such analysis helps us direct our efforts in most fruitful directions. We let each sample run for 20 minutes, first in isolation and then with a Random Impersonator. 65% or 390 of these attempt to reach a remote host. Table 2 shows the number of flows generated for TCP vs UDP, and Tables 3 and 4 show the number of flows per a given service port. HTTP and DNS account for majority of these flows. We then look deeper into HTTP flows, using our Symbolic Execution Engine to find out the purpose of each flow. The results are shown in Table 5. Majority of these flows serve C & C purpose or attempt to download a binary. These preliminary results demonstrate that there is a limited number of malware communication patterns that should be amenable to semi-manual or automated analysis needed to build Custom Impersonators for each pattern.

2.2.4 Fine-grained Firewall

Our four-step approach is realized by a Fine-grained Firewall. Similar to prior work [16] there are five possible actions for each malware communication: drop, allow, rate-limit, redirect or rewrite [16]. In our frame-

Port	# Flows	Purpose
80	32926	HTTP
9003	15818	SANS
8000	8149	non standard IRC, also HTTP
8003	7181	HTTP alternative
1608	2839	Smart Corp. License Manager
10021	612	Special Protocol
447	240	File distribution Management

Table 3: TCP flows, breakdown per port, showing ports with more than 100 requests

Port	# Flows	Purpose
53	18098	UDP
16471	42	Reach Master
P2P PORTS	660	

Table 4: UDP flows, breakdown per port

work we introduce two innovations into the containment mechanism: (1) We perform redirection not to a generic server, but to a Smart Impersonator. (2) We do not choose a single containment action for an entire connection. Instead, we can apply multiple containment actions at different points in a connection's history. For example, we may *allow* the initial handshake on a connection, then *rate-limit* the next few packets and then *drop* the rest. This flexible response is needed because it is impossible to tell the malware's intent at the very beginning of each connection. Through observations and interactions with the malware during a connection we can better understand the purpose of the communication and revise our decisions to better manage the risk.

For enforcement of policy decisions, we use a Fine-Grained Firewall, able to police malware traffic at a semantic level. This means that the Firewall can understand application-level headers and content in malware messages, as well as the header-level information found in individual packets, and to apply drop, rewrite, rate-limit, forward or redirect-to-Impersonator actions on the packets, flows and parts of flows. Two functionalities are needed for the Fine-Grained Firewall: (1) an expressive policy language at a high semantic level, and (2) a Deep Packet Inspection (DPI) firewall that efficiently examines entire packets. While we have not yet built the Fine-Grained Firewall, components to satisfy each of these functionalities already exist. in the open-source community We plan to modify and combine them to achieve the full functionality we require. Specifically, the Bro IDS language [20] is expressive enough to define signatures that relate to packet headers and contents, as well as to reassemble packet streams and perform state full signature matching, but it does not itself sit inline, and thus cannot act as a firewall. IP tables/Netfilter [15] can manipulate packets and flows efficiently but its policy language is not sufficiently expressive for our needs. In our work we plan to combine Netfilter's packet capture with Bro

Purpose	# requests
Downloading binary	236
Machine Registration	7
Contacting masters	2513
Non-standard HTTP	30
Connectivity Test	5

Table 5: HTTP breakdown

IDS's packet reassembly and matching, to create a novel firewall that can reason about packets, connections, applications and content and can take versatile actions on each of these targets inline.

2.2.5 Risk Management

The fourth step in our malware management is letting out communications that we could not successfully mimic inside the testbed and that were necessary for malware execution. For such communications we could not build the Custom Impersonator, because the STP solver could not solve the constraints we gave it. This will occur when the network input goes through non-linear transformations in the code, e.g., because it is a decryption key expected from the bot-master. Another possibility may be that we build the Custom Impersonator but the malware still does not execute.

In these cases, for useful experimentation malware communication must be allowed to leave the testbed. Since we cannot establish with high certainty the purpose of this communication we must resort to close monitoring of it and its effects on the remote host, so these rare communication channels cannot be misused to do harm.

We perform this monitoring in the following way. We detect TCP scans by monitoring the number of half-open TCP connections from the malware sample over time. Similarly UDP scans are detected by monitoring the ratio of successful (reply received from server) vs unsuccessful (ICMP unreachable, ICMP service unavailable or timeout) communication attempts. Very low thresholds can be used to detect and stop these scan campaigns as most communication attempts will be handled by an Impersonator and will not be affected by these thresholds. DoS attacks can be detected by observing persistent communication attempts from the malware with a destination that fails to generate sufficient replies (e.g. as done by D-WARD [30]). Spam campaigns can also be detected by setting a low threshold to the number of allowed e-mail messages. Known exploits should never be allowed to leave the testbed. We detect them by forcing all communication out to the Internet to pass through one or more IDS engines, such as Snort [22] to detect known exploits, and Bro [20], to detect protocol violations and malformed headers. These IDS engines are regularly updated with new malware signatures. While this cannot completely prevent our framework from spreading exploits, these checks ensure that only zero-day or

Project	Transparency	Communication	Strict Containment	Automaticity
Mesocosms [5]	No	No	Yes	Yes
GQ [16]	No	Yes	Yes	No
Botlab [13]	Yes	Yes	Yes	No
Deterlab [24]	No	Yes	Yes	No
Anubis [6]	No	No	Yes	Yes
MalwareLab [3]	Yes	No	-	Yes

Table 6: Comparison with Previous Work

polymorphic exploits escape. Zero-day exploits are extremely rare (only 18 were identified between 2008 and 2011 [10]), and thus the probability of us sourcing those is minimal.

3 Related Work

While there has been extensive work and advancements in malware analysis, there has been little done on automatic frameworks—frameworks which can enable safe malware experimentation without extensive input from specialized experts. Related work falls into one of the following categories: (1) full framework for experimentation, and (2) single target solutions, e.g un-packing, multi-path exploration etc. We only discuss solutions that propose full frameworks for malware experimentation as they are most closely related to our work.

Table 6 summarizes related frameworks [5, 16, 13, 24, 6, 3] and the challenges these frameworks address. *Hi-Fidelity Emulation:* column denotes if the framework provides solutions for anti-VM malware checks. *Handle-communication* column denotes if the framework allows *any* communication between malware and the Internet. *Strict Containment:* column denotes if strict containment measures are taken so that environment can guarantee no (or limited) risk to the Internet. *Automaticity:* denotes if human expert involvement is required for strict containment.

Mesocosms [5] authors implement a completely contained testbed, integrating a set of services required by botnets, such as IRC and DynDNS. While some of their work may be useful for developing our Smart Impersonators, their services offer one generic profile per service, while we plan to accommodate many profiles, each specific to one public server. Mesocosms does not require a human expert in the loop but it also does not allow any malware communication with the outside, which limits research utility.

GQ [16] and Botlab [13] allow limited communication with the outside world. GQ work proposes a malware execution farm that introduces containment as a first-order component in support of malware analysis. Similarly, Botlab—a real-time spam botnet monitoring system—also allows controlled communication with the Internet. But both GQ and Botlab require a human expert in the loop for strict containment while we do not.

The DeterLab testbed [4] has tried in the past to support live malware experimentation in a limited manner, through the T1/T2 framework [24]. The framework enables DeterLab users to allow traffic to certain outside destination IPs and ports. Only researchers that were vetted by the DeterLab's executive team had access to the framework, since this crude containment relied heavily on researcher expertise. This greatly differs from our proposed work where policies are mostly set automatically and risk to the Internet can be quantified.

Anubis [6] is a public service for analyzing malware. Samples are run in a sand-boxed environment for a certain amount of time after which a report is generated. No communication is allowed with the Internet.

Malwarelab[3], is an isolated environment built to test exploit kits' resiliency against software updates over time. It does not allow any communication with the Internet.

4 Conclusions and discussion

In this paper we advocated for better accessibility in live malware experimentation. We proposed a framework for automated, safe and public live-malware experimentation with two new advancements for high-fidelity transparent emulation and fine-grained automatic containment. We argue that a public framework benefits inexperienced and experienced users alike. These advancements take the burden off expert manual intervention and allow for easier setup, access and scaling in live malware analysis.

Opening up live malware experimentation using a shared environment will evolve malware research efforts and best practices at a much quicker pace, since sharing results and best practices does not require the effort of translating between differing experimental environments. Additionally, a publicly accessible environment provides easier access for new users who may bring expertise from other domains.

Through our fine-grained and automated policies, we expect that our proposed framework will be flexible and scalable enough to address a range of research needs and users. While we are at the early stages of development and exploration, we expect that feedback from the community will play a vital roll on the path to public and automated malware testbeds. We encourage discussion on enabling easier access to live malware experimentation and the ethical and technical challenges involved in

doing so. With feedback from the community, and thorough investigation, we hope to reach our goal of providing publicly accessible, safe and automated live malware experimentation.

References

[1] OpenVZ. http://openvz.org/.

[2] STP. http://people.csail.mit.edu/vganesh/STP_files/stp.html.

[3] ALLODI, L., KOTOV, V., AND MASSACCI, F. Malwarelab: Experimentation with cybercrime attack tools. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test* (Berkeley, CA, 2013), USENIX.

[4] BAJCSY, R., BENZEL, T., BISHOP, M., ET AL. Cyber Defense Technology Networking and Evaluation. *Commun. ACM 47*, 3 (2004).

[5] BARFORD, P., AND BLODGETT, M. Toward Botnet Mesocosms. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets* (2007).

[6] BAYER, U., HABIBI, I., BALZAROTTI, D., KIRDA, E., AND KRUEGEL, C. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)* (2009).

[7] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005).

[8] DEVINE, S. W., BUGNION, E., AND ROSENBLUM, M. Virtualization System including a Virtual Machine Monitor for a Computer with a Segmented Achitecture. *US Patent 6397242* (1998).

[9] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware Analysis via Hardware virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008).

[10] GOODIN, D. Zero-day attacks are meaner, more rampant than we ever thought. http://arstechnica.com/security/2012/10/zero-day-attacks-are-meaner-and-more-plentiful-than-thought/, Oct. 2012.

[11] HENDERSON, A., PRAKASH, A., YAN, L. K., HU, X., WANG, X., ZHOU, R., AND YIN, H. "make it work, make it right, make it fast", building a platform-neutral whole-system dynamic binary analysis platform. In *the International Symposium on Software Testing and Analysis* (San Jose, CA, 2014), ISSTA'14, ISSTA.

[12] INTEL. Intel 64 and IA-32 Architectures Software Developers Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-d\eveloper-manuals.html.

[13] JOHN, J. P., MOSHCHUK, A., GRIBBLE, S. D., AND KRISHNAMURTHY, A. Studying Spamming Botnets Using Botlab. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009).

[14] KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. Emulating Emulation-resistant Malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security* (2009).

[15] KIL, M. W., KIM, S. K., LEE, G., AND KWON, Y. A Development of Intrusion Detection and Protection System using Netfilter Framework. In *Proceedings of the 10th International Conference on Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing* (2005).

[16] KREIBICH, C., WEAVER, N., KANICH, C., CUI, W., AND PAXSON, V. GQ: Practical Containment for Measuring Modern Malware Systems. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (2011).

[17] LAWTON, K. P. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal 1996*, 29es.

[18] MARTIGNONI, L., PALEARI, R., FRESI ROGLIA, G., AND BRUSCHI, D. Testing System Virtual Machines. In *ISSTA* (2010).

[19] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing CPU Emulators. In *ISSTA* (2009).

[20] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th Conference on USENIX Security Symposium* (1998).

[21] PRATT, I., FRASER, K., HAND, S., LIMPACH, C., WARFIELD, A., MAGENHEIMER, D., NAKAJIMA, J., AND MALLICK, A. Xen 3.0 and the Art of Virtualization. In *Linux Symposium* (2005).

[22] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX conference on System administration* (1999).

[23] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (2008), ICISS '08, pp. 1–25.

[24] WROCLAWSKI, J., MIRKOVIC, J., FABER, T., AND SCHWAB, S. A Two-constraint Approach to Risky Cybersecurity Experiment Management. In *Invited paper at the Sarnoff Symposium* (2008).