Embedded Honeypotting

Frederico Araujo and Kevin W. Hamlen

Abstract *Language-based software cyber deception* leverages the science of compiler and programming language theory to equip software products with deceptive capabilities that misdirect and disinform attackers. A flagship example of software cyber deception is *embedded honeypots*, which arm live, commodity server software with deceptive attack-response and disinformation capabilities. This chapter presents a language-based approach to embedded honeypot design and implementation. Implications related to software architecture, compiler design, program analysis, and programming language semantics are discussed.

1 Introduction to Software Cyber Deception

Throughout the history of warfare, obfuscation and deception have been widely recognized as important tools for leveling the ubiquitous asymmetry between offensive and defensive combatants. In the modern era of cyber warfare, this asymmetry has perhaps never been more extreme. Despite a meteoric rise in worldwide spending on conventional cyber defensive technologies and personnel, the success rate and financial damage resulting from cyber attacks against software systems has escalated even faster, rising dramatically over the past few decades. The challenges can be traced in part to the inherently uneven terrain of the cyberspace battlefield—which typically favors attackers, who can wreak havoc by finding a single weakness, whereas defenders face the difficult task of protecting all possible vulnerabilities.

Frederico Araujo • Kevin W. Hamlen
The University of Texas at Dallas, 800 W. Campbell Rd. EC31, Richardson, TX, 75080-3021
e-mail: frederico.araujo@utdallas.edu,e-mail: hamlen@utdallas.edu

This research was supported in part by AFOSR award FA9550-14-1-0173, NSF awards #1054629 and #1027520, ONR award N00014-14-1-0030, and NSA award H98230-15-1-0271. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of the AFOSR, NSF, ONR, or NSA.

The attack surface exposed by the convergence of computing and networking poses particularly severe asymmetry challenges for defenders. Our computing systems are constantly under attack, yet the task of the adversary is greatly facilitated by information disclosed by the very defenses that respond to those attacks. This is because software and protocols have traditionally been conceived to provide informative feedback for error detection and correction, not to conceal the causes of faults. Many traditional security remediations therefore advertise themselves and their interventions in response to attempted intrusions, allowing attackers to easily map victim networks and diagnose system vulnerabilities. This enhances the chances of successful exploits and increases the attacker's confidence in stolen secrets or expected sabotage resulting from attacks.

Deceptive capabilities of software security defenses are an increasingly important, yet underutilized means to level such asymmetry. These capabilities mislead adversaries and degrade the effectiveness of their tactics, making successful exploits more difficult, time consuming, and cost prohibitive. Moreover, deceptive defense mechanisms entice attackers to make fictitious progress towards their malicious goals, gleaning important threat information all the while, without aborting the interaction as soon as an intrusion attempt is detected. This equips defenders with the ability to lure attackers into disclosing their actual intent, monitor their actions, and perform counterreconnaissance for attack attribution and threat intelligence gathering.

The deception stack. Deceptive techniques can be introduced at different layers of the software stack. Figure 1 illustrates this *deception stack* [31], which itemizes various deceptive capabilities available at the network, endpoint, application, and data layers. For example, computer networks known as *honeynets* [36, 37] intentionally purvey vulnerabilities that invite, detect, and monitor attackers; endpoint protection platforms [31] deceive malicious software by emulating diverse execution environments and creating fake processes at the application level to manipulate malware behavior; and falsified data can be strategically planted in decoy file systems to disinform and misdirect attackers away from high-value targets [38, 41].

The deception stack suggests that the difficulty of deceiving an advanced adversary increases as deceptions move up the stack. This is accurate if we compare, for instance, the work associated with emulating a network protocol with the challenge of crafting fake data that appear legitimate to the attacker: Protocols have clear and precise specifications and are therefore relatively easy to emulate, whereas there are many complex human factors influencing whether a specific datum is plausible and believable to a particular adversary.

In general, deceptive software defenses must employ one or more forms of deception, and leverage all layers of the deception stack to some degree in order to be effective against a persistent and skilled adversary. This chapter focuses on application-level deceptive techniques, which offer critical mediation capabilities between the network, endpoint and data deception layers. For example, an application-level, deception-enabled web server can ask the network-level firewall to allow certain payloads to reach the application layer, where it can then offer deceptive responses that misdirect the adversary into attacking decoy machines within the endpoint layer. These decoys can appeal to the data deception layer to purvey disinformation in the form of false secrets or even

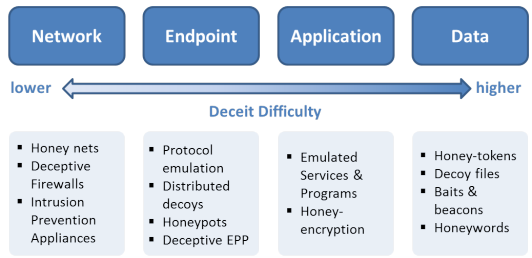


Fig. 1 Gartner's *deception stack* [31], with examples of deceptive technologies inhabiting each layer.

malware counter-attacks against adversaries. Such scenarios demonstrate the ongoing need for tools and techniques allowing organizations to engineer applications with proactive and deceptive capabilities that degrade attackers' methods and disrupt their reconnaissance efforts.

Towards this end, the remainder of this chapter introduces a language-based methodology for arming live, legacy server software with deceptive attack-response and disinformation capabilities. We refer to such capabilities as *embedded honeypots*. Embedded honeypots differ from traditional honeypots in that they reside within the actual, mission-critical software systems that attackers are seeking to penetrate; they are not independent decoy systems. Thus, embedded honeypots offer advanced deceptive remediations against informed adversaries who can identify and avoid traditional honeypots. In order to be adoptable, embedded honeypots imbue production server software with deceptive capabilities without degrading its performance or intended functionality. These new capabilities mislead advanced adversaries into wasting time and resources on phantom vulnerabilities and decoy file systems, and pave the way for an emerging science of *deception-facilitating software engineering*.

Honeypots [35] are information systems resources conceived to attract, detect, and gather attack information. Figure 2 presents an abbreviated timeline of the extensive history of honeypot research. A more comprehensive survey on recent advances and future trends in honeypot research can be found in [7]. Modern honeypots are usually classified according to the interaction level provided to potential attackers. *Low-interaction* honeypots [32] present a façade of emulated services without full server functionality, with the intent of detecting unauthorized activity via easily deployed pseudo-services. In contrast, *high-interaction* honeypots provide a relatively complete system with which attackers can interact, and are designed to capture detailed information on attacks. Despite their popularity, both low- and high-interaction honeypots are often detectable by informed adversaries (e.g., due to the limited services they purvey, or because they exhibit traffic patterns and data substantially different than genuine services). Embedded honeypots are highest-interaction in the sense that they purvey genuine services, and have access to genuine data, unlike traditional low- and high-interaction honeypots.

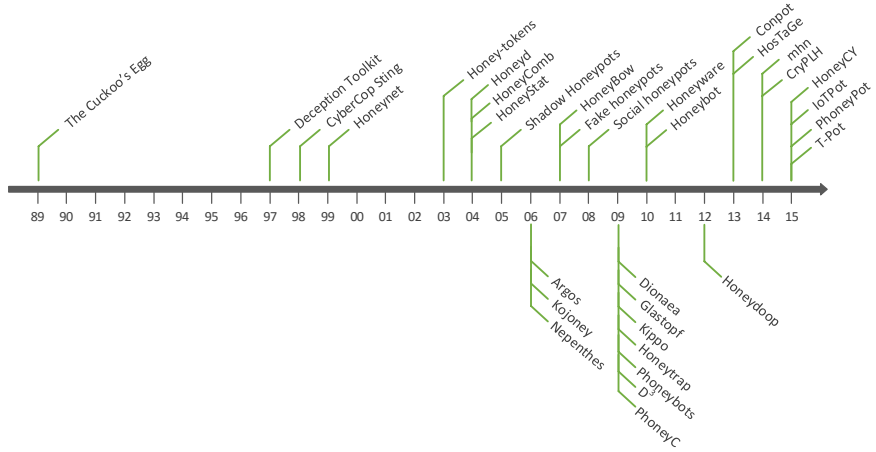


Fig. 2 Timeline of selected academic and industrial research on honeypots.

2 Honey-Patching: A New Software Cyber Deception Technology

When a software security vulnerability is discovered, the conventional defender reaction is to quickly *patch* the software to fix the problem. This standard reaction can backfire, however, if the patch has the side-effect of disclosing and highlighting other exploitable weaknesses in the defender's network. Unfortunately, such backfires are common; patches often behave in such a way that adversaries can reliably infer which systems have been patched, and therefore which are unpatched and vulnerable. The existence of at least some unpatched systems is almost inevitable, since patch adoption is rarely immediate—for example, testing is often required to ensure patch compatibility. Thus, most software security patches fix newly discovered vulnerabilities at the price of advertising to attackers which systems remain vulnerable. This has led to an adversarial culture for which *vulnerability probing* is a staple of the cyber killchain.

Cyber criminals easily probe today's Internet for vulnerable software, allowing them to focus their attacks on susceptible targets, in the following way. First, the attacker submits a malicious input (a *probe*) crafted to trigger a particular, known software bug in bulk to many servers across the network. Patched servers respond to the probe with a well-formed output, such as an error message; but unpatched servers behave erratically, such as by responding with a garbage string or crashing and restarting. Upon observing the latter response, the attacker next submits a more constructive malicious input to the unpatched servers, such as one that exploits the bug to hijack the victim software's control-flow, causing it to perform malicious actions on behalf of the attacker rather than merely crashing.

Honey-patching [5] is a game-changing alternative approach for anticipating and foiling these directed cyber attacks. The goal is to patch newly discovered software security vulnerabilities in such a way that future attempted exploits of the patched

vulnerabilities appear successful to attackers even when they are not. This masks patching lapses, impeding attackers from easily discerning which systems are genuinely vulnerable and which are actually patched systems masquerading as unpatched systems. Detected attacks are transparently redirected to isolated, unpatched decoy environments that possess the full interactive power of the targeted victim server, but that disinform adversaries with honey-data and aggressively monitor adversarial behavior.

Deceptive honey-patching capabilities thereby constitute an advanced, language-based, active defense technique that can impede, confound, and misdirect attacks, and significantly raise attacker risk and uncertainty. In addition to helping protect networks where honey-patches are deployed, the practice also contributes to the public cyber welfare: Once a honey-patch for a particular software vulnerability has been adopted by some, attacks against all networks become riskier for attackers. This is because attackers can no longer reliably identify all the vulnerable systems and determine where to focus their attacks, or even assert whether they are gathering genuine data. Any ostensibly vulnerable network could be a honey-patch in disguise, and any exfiltrated secret could potentially be disinformation.

Threat model. Honey-patches add a layer of deception to confound exploits of known (patchable) vulnerabilities, which constitute the majority of exploited vulnerabilities in the wild. Previously unknown (i.e., zero-day) exploits remain potential threats, since such vulnerabilities are typically neither patched nor honey-patched. However, even zero-days can be potentially mitigated through cooperation of honey-patches with other layers of the deception stack. For example, a honey-patch that collects identifying information about a particular adversary seeking to exploit a known vulnerability can convey that collected information to a network-level intrusion detection system, which can then potentially identify the same adversary seeking to exploit a previously unknown vulnerability.

Although honey-patches primarily mitigate exploits of known vulnerabilities, they can effectively mitigate exploits whose attack payloads might be completely unique and therefore unknown to defenders. Such payloads might elude network-level monitors, and are therefore best detected at the software level at the point of exploit. Attackers might also use one payload for reconnaissance but reserve another for the final attack. Misleading the attacker into launching the final attack is therefore useful for discovering the final attack payload, which can divulge attacker strategies and goals not discernible from the reconnaissance payload alone.

Honey-patching is typically used in conjunction with standard access control protections, such as process isolation and least privilege. Attacker requests are therefore typically processed by a server possessing strictly user-level privileges, and must therefore leverage web server bugs and kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing other users' memory to access confidential data. The defender's ability to thwart these and future attacks stems from his ability to deflect attackers to fully isolated decoys and perform counterreconnaissance (e.g., attack attribution and information gathering).

2.1 Honey-patch Design Principles

Although the honey-patching concept is fairly straightforward, many significant security and performance challenges must be surmounted to realize it in practice. For example, a honey-patch that naïvely forks the entire server process to create a decoy clone process in response to attempted intrusions, inadvertently copies any secrets in the victim process's address space, such as encryption keys of concurrent sessions, over to the child decoy. Such an approach would be disastrous in practice, since the attack is allowed to succeed in the decoy, thereby giving the attacker potential access to secrets it may contain.

Moreover, practical adoption requires that honey-patches (1) introduce almost no overhead for legitimate users, (2) perform well enough for attackers that attack failures are not placarded, and (3) offer high compatibility with software that boasts aggressive multi-processing, multi-threading, and active connection migration across IPs. Solutions must therefore be sufficiently modular and generic that administrators require only a superficial, high-level understanding of each patch's structure and semantics to reformulate it as an effective honey-patch.

Specifically, effective honey-patching requires that remote forking of attacker sessions to decoys must happen live, with no perceptible disruption in the target application. This means that established connections—in particular, the attacker's connection—must not be broken. In addition, decoy deployment must be fast, to avoid offering overt, reliable timing channels that advertise the honey-patch. Finally, all sensitive data must be redacted before the decoy resumes execution to avoid giving the attacker potential access to user secrets.

Together, these requirements motivate three main design decisions: First, the required time performance precludes system-level cloning (e.g., VM cloning [9]) for session forking; instead, a lighter-weight, finer-grained alternative based on process migration through *checkpoint-restart* [25] is recommended. To scale to many concurrent attacks, *OS-level virtualization* should be leveraged to deploy forked processes to decoy containers, which can be created, deployed, and destroyed orders of magnitude faster than other virtualization techniques, such as full virtualization or para-virtualization [40].

OS-level virtualization allows multiple guest nodes (*containers*) to share the kernel of their controlling host. Linux containers (LXC) [23] implement OS-level virtualization, with resource management via process control groups and full resource isolation via Linux namespaces. This ensures that each container's processes, file system, network, and users remain mutually isolated. Fine-grained control of resource utilization prevents any container from starving its host. Furthermore, LXC supports containers backed by *overlayfs* snapshots, which is key for efficient container management and fast decoy deployment.

Second, transparent redirection of attacker sessions can be accomplished via a fine-grained, lightweight process migration technique based on checkpoint-restart, which facilitates the live migration of attacker sessions to decoys. This approach benefits from the synergy between mainstream Linux kernel APIs and user-space tools, allowing for a

small freezing time of the target application and a special relocation mechanism for established connections that allows for transparent session migration.

Process migration through checkpoint-restart is the act of transferring a running process between two nodes by dumping its state on the source and resuming its execution on the destination. This problem is especially relevant for high-performance computing [15, 39]. As a result, several tools have been developed to support performance-critical process checkpoint-restart (e.g., BLCR [14], DMTCP [2], and CRIU [11]). Process checkpoint-restart plays a pivotal role in making the honey-patch concept viable. It provides a fast and seamless mechanism to enable transparent forking of attacker sessions, and scales well even in small environments due to its process-level granularity, which reduces the overall resources required to migrate the attacker process.

Third, to guarantee that successful exploits do not afford attackers access to sensitive data stored in application memory, honey-patches should implement a dynamic *secret redaction* mechanism that redacts the attacker process image during forking. This censors sensitive data from process memory before the forked (unpatched) session resumes. Forked decoys host a deceptive file system that omits all secrets, and that can be laced with disinformation to further deceive, delay, and misdirect attackers.

2.2 Architecture

The REDHERRING architecture [5], depicted in Figure 3, embodies these design decisions by using process-level cloning and OS-level virtualization to achieve lightweight, resource-efficient, and fine-grained redirection of attacker sessions to sandboxed decoy environments in which secrets have been redacted with honey-data. Within this framework, developers use honey-patches to provide the same level of security as conventional patches, yet have the additional ability to deceive attackers.

Central to the system is a *reverse proxy* that acts as a transparent proxy between users and internal servers deployed as LXC containers. The *target* container hosts the honey-patched web server instance, and the n decoys form the pool of ephemeral containers managed by the *LXC Controller*. The decoys serve as temporary environments for attacker sessions. Each container runs a *CR-Service* (Checkpoint/Restore) daemon, which exposes an interface controlled by the *CR-Controller* for remote checkpoint and restore.

Honey-patching API. To achieve low-coupling between target application and honey-patching logic, the honey-patch mechanism can be realized as a tiny library (e.g., implemented as a dynamically loadable C library). The library exposes three core API functions:

- `hp_init` ($pgid, pid, tid, sk$): initialize honey-patch with the process group $pgid$, process pid, thread tid, and socket descriptor sk of the session.

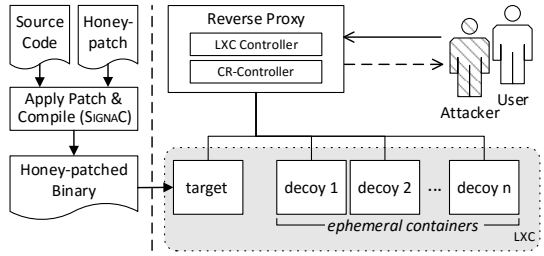


Fig. 3 REDHERRING system architecture overview

- `hp_fork()`: initiate the attacker session remote forking process, implementing the honey-patching core logic.
- `hp_skip(`c`)`: skip over block c if in a decoy.

Function `hp_init` initializes the honey-patch with the necessary information to handle subsequent session termination and resurrection. It is invoked once per connection, at the start of the session life cycle. For example, in the Apache web server, this immediately follows acceptance of an HTTP request and handing the newly created session off to a child process or worker thread; in Lighttpd and Nginx web servers, it follows the accept event for new connections.

Lighttpd [22] and Nginx [27] are web servers whose designs are significantly different from Apache [3]. The most notable difference lies in the processing model of these servers, which employs non-blocking systems calls (e.g., select, poll, epoll) to perform asynchronous I/O operations for concurrent processing of multiple HTTP requests. In contrast, Apache dispatches each request to a child process or thread [30]. The ability to add deception capabilities to these three very different server architectures evidences the versatility of honey-patching as a software defense paradigm.

Listing 1 details the basic steps of `hp_fork`. Line 3 determines the application context, which can be either `target` (denoting the target container) or `decoy`. In a decoy, the function does nothing, allowing multiple exploits within a single attacker session to continue within the same decoy. In the target, a fork is initiated, consisting of four steps: (1) Line 5 registers the signal handler for session termination and resurrection. (2) Line 6 sends a *fork request* containing the attacker session's $pgid$, pid, and tid to the proxy's CR-Controller. (3) Line 7 synchronizes checkpoint and restore of the attacker session in the target and decoy, respectively, and guarantees that sensitive data is redacted from memory before the clone is allowed to resume. (4) Once forking is complete and the attacker session has been resurrected, the honey-patch context is saved and the attacker session resumes in the decoy.

The fork request (step 2) achieves high efficiency by first issuing a system `fork` to create a shallow, local clone of the web server process. This allows event-driven web servers to continue while attacker sessions are forked onto decoys, without interrupting the main event-loop. It also lifts the burden of synchronizing concurrent checkpoint

Listing 1 `hp_fork function`

```
1 void hp_fork()
2 {
3   read_context();          //read context (target/decoy)
4   if (decoy) return;       //if in decoy, do nothing
5   register_handler();      //register signal handler
6   request_fork();          //fork session to decoy
7   wait();                  //wait until fork process has finished
8   save_context();          //save context and resume
9 }
```

operations, since CRIU injects a Binary, Large OBject (BLOB) into the target process memory space to extract state data during checkpoint.

The context-sensitivity of this framework allows the honey-patch code to exhibit context-specific behavior: In decoy contexts, `hp_skip` elides the execution of the code block passed as an argument to the macro, elegantly simulating the unpatched application code. In a target context, it is usually never reached due to the fork. However, if forking silently fails (e.g., due to resource exhaustion), it abandons the deception and conservatively executes the original patch's corrective action for safety.

LXC Pool. The decoys into which attacker sessions are forked are managed as a pool of Linux containers controlled by the LXC Controller. The controller exposes two operations to the proxy: *acquire* (to acquire a container from the pool), and *release* (to release back a container to the pool). Each container follows the life cycle depicted in Figure 4. Upon receiving a fork request, the proxy acquires the first available container from the pool. The acquired container holds an attacker session until (1) the session is deliberately closed by the attacker, (2) the connection's *keep-alive* timeout expires, (3) the ephemeral container crashes, or (4) a session timeout is reached. The last two conditions are common outcomes of successful exploits. In any of these cases, the container is released back to the pool and undergoes a recycling process before becoming available again.

Recycling a container encompasses three sequential operations: *destroy*, *clone* (which creates a new container from a template in which legitimate files are replaced with honeyfiles), and *start*. These steps happen swiftly for two main reasons. First, the lightweight virtualization implemented by LXC allows containers to be destroyed and started similarly to how OS processes are terminated and created. Second, ephemeral containers are deployed as overlayfs-based clones, making the cloning step almost instantaneous. The overlay file system is backed by a regular directory (the *template*) to clone new *overlayfs* containers (decoys), mounting the template's root file system as a read-only lower mount and a new private delta directory as a read-write upper mount. The template used to clone decoys is a copy of the target container in which all sensitive files are replaced with honey-files.

CR-Service. The Reverse Proxy uses the CR-Controller module to communicate with CR-Service daemons running in the background of each container. The CR-Service uses an extended version of CRIU (Checkpoint/Restore In Userspace) [11] to checkpoint attacker sessions on the target and restore them on decoys. Each CR-Service implements

a *façade* that exposes CR operations to the proxy's CR-Controller through a simple RPC protocol based on Protocol Buffers [16]. To enable fast, OS-local RPC communication between proxy and containers, IPC sockets (a.k.a., Unix domain sockets) are used.

However, because IPC sockets rely on the file system as an address namespace and the proxy runs on the host, establishing cross-container connections becomes difficult: host and containers file-systems are opaque to each other (due to namespace isolation). To overcome this issue, a directory located in the host is configured as bind mount to all containers to be used as bridge for IPC sockets, thus enabling the establishment of IPC connections between the CR-Controller running in the host and the CR-Service instances running inside containers.

Reverse Proxy. The proxy plays a dual role in the honey-patching system, acting as (1) a *transport layer transparent proxy*, and (2) an *orchestrator* for attacker session forking.

As a transparent proxy, its main purpose is to hide the backend web servers and route client requests. To serve each client's request, the proxy server accepts a downstream socket connection from the client and binds an upstream socket connection to the backend server, allowing application-layer sessions to be processed transparently between the client and the backend server. To keep its size small, the proxy neither manipulates message payloads, nor implements any rules for detecting attacks. There is also no session caching. This makes it extremely innocuous and lightweight. The proxy is implemented as a transport-layer reverse proxy to reduce routing overhead and support the variety of protocols operating above TCP, including SSL/TLS.

As an orchestrator, the proxy listens for fork requests and coordinates the attacker session forking as shown in Figure 5. Under legitimate load, the proxy simply routes user requests to the target and routes server responses to users. However, attack inputs elicit the following alternate workflow:

Step 1: The attacker probes the server with a crafted request (denoted by request `GET /malicious` in Figure 5).

Step 2: The reverse proxy transparently routes the request to the backend target web server.

Step 3: The request triggers the honey-patch (i.e., when the honey-patch detects an attempted exploit of the patched vulnerability) and issues a fork request to the reverse proxy.

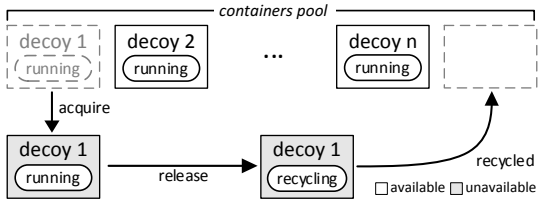


Fig. 4 Linux containers pool and decoys life cycle

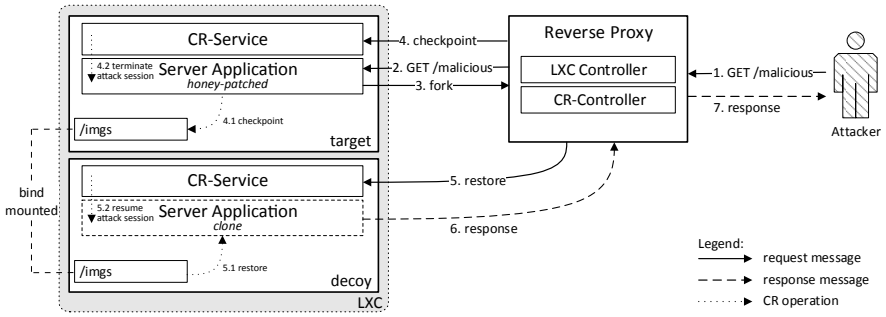


Fig. 5 Attacker session forking. Numbers indicate the sequential steps taken to fork an attacker session.

Step 4: The proxy's CR-Controller processes the request, acquires a decoy from the LXC Pool, and issues a checkpoint RPC request to the target's CR-Service. The CR-Service

4.1: checkpoints the running web server instance to the `/imgs` directory; and
4.2: signals the attacker session with a termination code, gracefully terminating it.

Step 5: Upon checkpoint completion, the CR-Controller commands the decoy's CR-Service to restore the dumped web server images on the decoy. The CR-Service then

5.1: restores a clone of the web server from the dump images located in the `/imgs` directory; and
5.2: signals the attacker session with a resume code, and cleans the dump data from `/imgs`.

Step 6: The attacker session resumes on the decoy, and a response is sent back to the reverse proxy.

Step 7: The reverse proxy routes the response to the attacker.

Throughout this workflow, the attacker's session forking is completely transparent to the attacker. To avoid any substantial overhead for transferring files between target and decoys, each decoy's `/imgs` folder is bind-mounted to the target's `/imgs` directory. After the session has been forked to the decoy, it behaves like an unpatched server, making it appear that no redirection has taken place and the original probed server is vulnerable.

Established TCP Connection Relocation. Target and decoys are fully isolated containers running on separate namespaces. As a result, each container is assigned a unique IP in the internal network, which affects how active connections are moved from the target to a decoy. To realize this use case, an extension to CRIU is implemented as part of the honey-patching framework to support relocation of TCP connections during process restoration. In what follows, we will discuss important details of the implementation.

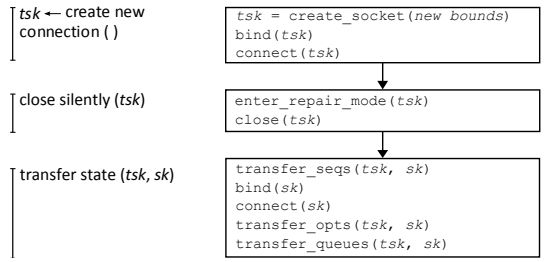


Fig. 6 Procedure for TCP connection relocation

The reverse proxy always routes legitimate user connections to the target; hence, there is no need to restore the state of connections for these users when restoring the web server on a decoy. Legitimate connections can simply be restored to *drainer sockets*, since we have no interest in maintaining legitimate user interaction with the decoys. This ensures that the associated user sessions are restored to completion without interrupting the overall application restoration.

Conversely, the attacker connection must be restored to its dumped state when switching the attacker session to a decoy. This is important to avoid connection disruption and to allow transparent session migration (from the perspective of the attacker). To accomplish this, the proxy dynamically establishes a new backend TCP connection between proxy and decoy containers in order to hold the attacker session communication. Moreover, a mechanism based on *TCP repair options* [10] is employed to transfer the state of the original attacker's session socket (bound to the target IP address) into the newly created socket (bound to the decoy IP address).

Figure 6 describes the connection relocation mechanism, implemented as a step of the attacker's session restore process. At process checkpoint, the state information of the original socket `sk` is dumped together with the process image (not shown in the figure). This includes connection bounds, previously negotiated socket options, sequence numbers, receiving and sending queues, and connection state. During process restore, the connection is relocated to the assigned decoy by (1) connecting a new socket `tsk` to the proxy `$port` given in the restore request, (2) setting `tsk` to *repair mode* and silently closing the socket (i.e., no `FIN` or `RST` packages are sent to the remote end), and (3) transferring the connection state from `sk` to `tsk` in repair mode. Once the new socket `tsk` is handed over to the restored attacker session, the relocation process has completed and communication resumes, often with an HTTP response being sent back to the attacker.

3 Process Image Secret Redaction

To more effectively enforce data confidentiality and privacy concerns in security-sensitive, deceptive software environments, honey-patching leverages new compiler techniques that equip software with dynamic secret redaction capabilities. The resulting software responds to emerging cyber attacks by quickly and comprehensively substituting all

secrets in its address space with honey-tokens that disinform attackers. Safe, efficient redaction of secrets from program address spaces has many applications, including the safe release of program memory dumps to software developers for debugging purposes, mitigation of cyber-attacks via runtime self-censoring in response to intrusions, and attacker deception through honeypotting.

Realizing such runtime process secret redaction in practice elicits at least two significant challenges. First, the redaction step must yield a runnable program process. Non-secrets must therefore not be conservatively redacted, lest data critical for continuing the program's execution be deleted. Secret redaction for running processes is hence especially sensitive to *label creep* and *over-tainting* failures.

Label creep and over-tainting are classic challenges in the data confidentiality enforcement literature. The former refers to the tendency of a datum's classification level to become ever more restrictive over its lifetime, until even those who own the data may lack the privileges to access it. The latter refers to the tendency of information security systems to over-classify even non-secrets as confidential—for example, due to non-secrets coming into brief contact with confidential data during information processing.

Second, many real-world programs targeted by cyber-attacks were not originally designed with information flow tracking support, and are often expressed in low-level, type-unsafe languages, such as C/C++. A suitable solution must be amenable to retrofitting such low-level, legacy software with annotations sufficient to distinguish non-secrets from secrets, and with efficient flow-tracking logic that does not impair performance.

This section summarizes a suitable dynamic secret redaction solution [4] that has been integrated into the LLVM compiler's [21] DataFlow Sanatizer (DFSan) infrastructure [13], which adds byte-granularity taint-tracking support to C/C++ programs at compile-time. At the source level, DFSan's taint-tracking capabilities are purveyed as runtime data-classification, data-declassification, and taint-checking operations, which programmers add to their programs to identify secrets and curtail their flow at runtime. Unfortunately, straightforward use of this interface for redaction of large, complex legacy codes can lead to severe over-tainting, or requires an unreasonably detailed retooling of the code with copious classification operations. This is unsafe, since missing even one of these classification points during retooling risks disclosing secrets to adversaries.

To overcome these deficiencies, our research has augmented LLVM with a declarative, type annotation-based secret-labeling mechanism for easier secret identification, and a new label propagation semantics, called *Pointer Conditional-Combine Semantics* (PC^2S). The semantics efficiently distinguishes secret data within C-style graph data structures from the non-secret structure that houses the data. This partitioning of the bytes greatly reduces over-tainting and the programmer's annotation burden, and proves critical for precisely and efficiently redacting secret process data whilst preserving process operation after redaction.

Listing 2 Apache's URI parser function (excerpt)

```
1  /* first colon delimits username:password */
2  s1 = memchr(hostinfo, ':', s - hostinfo);
3  if (s1) {
4   uptr->user = apr_pstrmemdup(p, hostinfo, s1-hostinfo);
5   ++s1;
6   uptr->password = apr_pstrmemdup(p, s1, s - s1);
7  }
```

Listing 3 Apache's session record (excerpt)

```
1  typedef struct {
2      NONSECRET apr_pool_t *pool;
3      NONSECRET apr_uuid_t *uuid;
4      SECRET_STR const char *remote_user;
5      apr_table_t *entries;
6      ...
7  } SECRET session_rec;
```

3.1 Sourcing & Tracking Secrets

Taint-tracking conceptually entails labeling each byte of process memory with a security label that denotes its classification level. At compile-time, a taint-tracking compiler instruments the resulting object code with extra code that propagates these labels alongside the data they label.

Extending such taint-tracking to low-level, legacy code not designed with taint-tracking in mind is often difficult. For example, the standard approach of specifying taint introductions as annotated program inputs often proves too coarse for inputs comprising low-level, unstructured data streams, such as network sockets. Listing 2 exemplifies the problem using a code excerpt from the Apache web server [3]. The excerpt partitions a byte stream (stored in buffer `s1`) into a non-secret user name and a secret password, delimited by a colon character. Naïvely labeling input `s1` as secret to secure the password causes the compiler to over-taint the user name (and the colon delimiter, and the rest of the stream), leading to excessive over-tainting—everything associated with the stream becomes secret, with the result that nothing can be safely divulged.

A correct solution must more precisely identify data field `uptr->password` (but not `uptr->user`) as secret after the unstructured data has been parsed. This is achieved in DFSan by manually inserting a runtime classification operation after line 6. However, on a larger scale this brute-force labeling strategy imposes a dangerously heavy annotation burden on developers, who must manually locate all such classification points. In C/C++ programs littered with pointer arithmetic, the correct classification points can often be obscure. Inadvertently omitting even one classification risks information leaks.

To ease this burden, a better solution is to introduce a mechanism whereby developers can identify secret-storing structures and fields *declaratively* rather than operationally. For example, to correctly label the password in Listing 2 as secret, users may add type qualifier `SECRET_STR` to the password field's declaration in its abstract datatype definition. A modified LLVM compiler responds to this static annotation by instrumenting the program with instructions that dynamically taint all values assigned to the password field. Since datatypes typically have a single point of definition (in contrast to the many code points that access them), this greatly reduces the annotation burden imposed upon code maintainers.

In cases where the appropriate taint is not statically known (e.g., if each password requires a different, user-specific taint label), parameterized type-qualifier `SECRET`$\langle f \rangle$ identifies a function f that computes the appropriate taint label at runtime.

Unlike traditional taint introduction semantics, which label program input values and sources with taints, recognizing structure fields as taint sources requires a new form of taint semantics that conceptually interprets dynamically identified *memory addresses* as taint sources. For example, a program that assigns address `&(uptr->password)` to pointer variable p, and then assigns a freshly allocated memory address to $*p$, must automatically identify the freshly allocated memory as a new taint source, and thereafter taint any values stored at $*p[i]$ (for all indexes i).

To achieve this, DFSan's *pointer-combine semantics (PCS)* feature is extended to optionally combine (i.e., join) the taints of pointers and pointees during pointer dereferences. Specifically, when *PCS on-load* is enabled, read-operation $*p$ yields a value tainted with the join of pointer p's taint and the taint of the value to which p points; and when *PCS on-store* is enabled, write-operation $*p := e$ taints the value stored into $*p$ with the join of p's and e's taints. Using PCS leads to a natural encoding of `SECRET` annotations as pointer taints. Continuing the previous example, PCS propagates `uptr->password`'s taint to p, and subsequent dereferencing assignments propagate the two pointers' taints to secrets stored at their destinations.

PCS works well when secrets are always separated from the structures that house them by a level of pointer indirection, as in the example above (where `uptr->password` is a pointer to the secret rather than the secret itself). However, label creep difficulties arise when structures mix secret values with non-secret pointers. To illustrate, consider a singly linked list ℓ of secret integers, where each integer has a different taint. In order for PCS on-store to correctly classify values stored to ℓ`->secret_int`, pointer ℓ must have taint γ_1, where γ_1 is the desired taint of the first integer. But this causes stores to ℓ`->next` to incorrectly propagate taint γ_1 to the node's next-pointer, which propagates γ_1 to subsequent nodes when dereferenced. In the worst case, all nodes become labeled with all taints. Such issues have spotlighted effective pointer tainting as a significant challenge in the taint-tracking literature [12, 19, 33, 34].

To address this shortcoming, PC^{2}S semantics generalize PCS semantics by augmenting them with pointer-combine *exemptions* conditional upon the static type of the pointee. In particular, a PC^{2}S taint-propagation policy may dictate that taint labels are not combined when the pointee has pointer type. Hence, ℓ`->secret_int` receives ℓ's taint because the assigned expression has integer type, whereas ℓ's taint is *not* propagated to ℓ`->next` because the latter's assigned expression has pointer type. Empirical evaluation shows that just a few strategically selected exemption rules expressed using this refined semantics suffices to vastly reduce label creep while correctly tracking all secrets in large legacy source codes.

In order to strike an acceptable balance between security and usability, the solution only automates tainting of C/C++ style structures whose non-pointer fields share a common taint. Non-pointer fields of mixed taintedness within a single struct are not supported automatically because C programs routinely use pointer arithmetic to reference multiple fields in a struct via a common pointer (imparting the pointer's taint to all the struct's non-pointer fields)—for example, when copying structures, or when

programs	$\mathcal{P} ::= \overline{c}$	*locations*	$\ell ::=$ memory addresses
commands	$c ::= v := e \mid \texttt{store}(\tau, e_1, e_2)$	*environment*	$\Delta : v \rightharpoonup u$
	$\mid \texttt{ret}(\tau,\ e) \mid \texttt{br}(e, e_1, e_0)$	*prog counter*	pc
	$\mid \texttt{call}(\tau, e, \overline{args})$	*stores*	$\sigma : (\ell \rightharpoonup u) \cup (v \rightharpoonup \ell)$
expressions	$e ::= v \mid \langle u, \gamma \rangle \mid \Diamond_b(\tau, e_1, e_2)$	*functions*	f
	$\mid \texttt{load}(\tau, e)$	*function table*	$\phi : f \rightharpoonup \ell$
binary ops	$\Diamond_b ::=$ typical binary operators	*taint contexts*	$\lambda : (\ell \cup v) \rightharpoonup \gamma$
variables	v	*propagation*	$\rho : \overline{\gamma} \to \gamma$
values	$u ::=$ values of underlying IR	*prop contexts*	$\mathcal{A} : f \to \rho$
types	$\tau ::= ptr\ \tau \mid \tau\ \overline{\tau} \mid$ primitive types	*call stack*	$\Xi ::= nil$
taint labels	$\gamma \in (\Gamma, \sqsubseteq)$ (label lattice)		$\mid \langle f,\ pc,\ \Delta,\ \overline{\gamma} \rangle :: \Xi$

Fig. 7 Intermediate representation syntax.

marshalling and demarshalling them to/from streams. This approach therefore targets the common case in which the taint policy is expressible at the granularity of structures, with exemptions for fields that point to other (differently tainted) structure instances. This corresponds to the usual scenario where a non-secret graph structure (e.g., a tree) stores secret data in its nodes.

With these new language extensions, users label structure datatypes as `SECRET` (implicitly introducing a taint to all fields within the structure), and additionally annotate pointer fields as `NONSECRET` to exempt their taints from pointer-combines during dereferences. Pointers to dynamic-length, null-terminated secrets get annotation `SECRET_STR`. For example, Listing 3 illustrates the annotation of `session_req`, used by Apache to store remote users' session data. Finer-granularity policies remain enforceable, but require manual instrumentation via DFSan's API, to precisely distinguish which of the code's pointer dereference operations propagate pointer taints. This solution thus complements existing approaches.

3.2 Formal Semantics

For explanatory precision, the new taint-tracking semantics is formally defined in terms of the simple, typed intermediate language (IL) in Figure 7, inspired by prior work [33]. The simplified IL abstracts irrelevant details of LLVM's IR language, capturing only those features needed to formalize the analysis.

Language Syntax. Programs $\mathcal{P}$ are lists of commands, denoted $\overline{c}$. Commands consist of variable assignments, pointer-dereferencing assignments (stores), conditional branches, function invocations, and function returns. Expressions evaluate to value-taint pairs $\langle u, \gamma \rangle$, where u ranges over typical value representations, and γ is the taint label associated with u. Labels denote sets of taints; they therefore comprise a lattice ordered

$$\frac{}{\sigma,\Delta,\lambda \vdash u \Downarrow \langle u,\bot\rangle}\ \textsc{Val} \qquad \frac{}{\sigma,\Delta,\lambda \vdash v \Downarrow \langle \Delta(v),\lambda(v)\rangle}\ \textsc{Var}$$

$$\frac{\sigma,\Delta,\lambda \vdash e_1 \Downarrow \langle u_1,\gamma_1\rangle \quad \sigma,\Delta,\lambda \vdash e_2 \Downarrow \langle u_2,\gamma_2\rangle}{\sigma,\Delta,\lambda \vdash \Diamond_b(\tau,\ e_1,\ e_2) \Downarrow \langle u_1 \Diamond_b u_2, \gamma_1 \sqcup \gamma_2\rangle}\ \textsc{BinOp}$$

$$\frac{\sigma,\Delta,\lambda \vdash e \Downarrow \langle u,\gamma\rangle}{\sigma,\Delta,\lambda \vdash \texttt{load}(\tau,\ e) \Downarrow \langle \sigma(u), \rho_{load}(\tau,\gamma,\lambda(u))\rangle}\ \textsc{Load}$$

$$\frac{\sigma,\Delta,\lambda \vdash e \Downarrow \langle u,\gamma\rangle \quad \Delta' = \Delta[v \mapsto u] \quad \lambda' = \lambda[v \mapsto \gamma]}{\langle \sigma,\Delta,\lambda,\Xi,pc, v := e\rangle \to_1 \langle \sigma,\Delta',\lambda',\Xi,pc+1,\mathcal{P}[pc+1]\rangle}\ \textsc{Assign}$$

$$\frac{\begin{array}{c}\sigma,\Delta,\lambda \vdash e_1 \Downarrow \langle u_1,\gamma_1\rangle \\ \sigma,\Delta,\lambda \vdash e_2 \Downarrow \langle u_2,\gamma_2\rangle \quad \sigma' = \sigma[u_1 \mapsto u_2] \quad \lambda' = \lambda[u_1 \mapsto \rho_{store}(\tau,\gamma_1,\gamma_2)]\end{array}}{\langle \sigma,\Delta,\lambda,\Xi,pc,\texttt{store}(\tau,e_1,e_2)\rangle \to_1 \langle \sigma',\Delta,\lambda',\Xi,pc+1,\mathcal{P}[pc+1]\rangle}\ \textsc{Store}$$

$$\frac{\sigma,\Delta,\lambda \vdash e \Downarrow \langle u,\gamma\rangle \qquad \sigma,\Delta,\lambda \vdash e_{(u\,?\,1:0)} \Downarrow \langle u',\gamma'\rangle}{\langle \sigma,\Delta,\lambda,\Xi,pc,\texttt{br}(e,e_1,e_0)\rangle \to_1 \langle \sigma,\Delta,\lambda,\Xi,u',\mathcal{P}[u']\rangle}\ \textsc{Cond}$$

$$\frac{\begin{array}{c}\sigma,\Delta,\lambda \vdash e_1 \Downarrow \langle u_1,\gamma_1\rangle \quad \cdots \quad \sigma,\Delta,\lambda \vdash e_n \Downarrow \langle u_n,\gamma_n\rangle \quad \Delta' = \Delta[\overline{params_f} \mapsto \overline{u_1 \cdots u_n}] \\ \lambda' = \lambda[\overline{params_f} \mapsto \overline{\gamma_1 \cdots \gamma_n}] \quad fr = \langle f, pc+1, \Delta, \overline{\gamma_1 \cdots \gamma_n}\rangle\end{array}}{\langle \sigma,\Delta,\lambda,\Xi,pc,\texttt{call}(\tau,f,\overline{e_1 \cdots e_n})\rangle \to_1 \langle \sigma,\Delta',\lambda',fr :: \Xi,\phi(f),\mathcal{P}[\phi(f)]\rangle}\ \textsc{Call}$$

$$\frac{\sigma,\Delta,\lambda \vdash e \Downarrow \langle u,\gamma\rangle \quad fr = \langle f, pc', \Delta', \overline{\gamma}\rangle \quad \lambda' = \lambda[v_{ret} \mapsto \mathcal{A}\, f\, \overline{\gamma}]}{\langle \sigma,\Delta,\lambda,fr :: \Xi,pc,\texttt{ret}(\tau,e)\rangle \to_1 \langle \sigma,\Delta'[v_{ret} \mapsto u],\lambda',\Xi,pc',\mathcal{P}[pc']\rangle}\ \textsc{Ret}$$

Fig. 8 Operational semantics of a generalized label propagation semantics.

NCS	$\rho_{\{load,store\}}(\tau,\gamma_1,\gamma_2) := \gamma_2$
PCS	$\rho_{\{load,store\}}(\tau,\gamma_1,\gamma_2) := \gamma_1 \sqcup \gamma_2$
PC^{2}S	$\rho_{\{load,store\}}(\tau,\gamma_1,\gamma_2) := (\tau \text{ is } ptr)\ ?\ \gamma_2 : (\gamma_1 \sqcup \gamma_2)$

Fig. 9 Polymorphic functions for no-combine, pointer-combine, and PC^{2}S propagation policies.

by subset ($\sqsubseteq$), with the empty set $\bot$ at the bottom (denoting public data), and the universe $\top$ of all taints at the top (denoting maximally secret data). Join operation $\sqcup$ denotes least upper bound (union) of taint sets.

Variable names range over identifiers and function names, and the type system supports pointer types, function types, and typical primitive types. Since DFSan's taint-tracking is dynamic, we here omit a formal static semantics and assume that programs are well-typed. Execution contexts are comprised of a store σ relating locations to values and variables to locations, an environment Δ mapping variables to values, and a tainting context λ mapping locations and variables to taint labels. Additionally, to express the semantics of label propagation for external function calls (e.g., runtime library API calls), function table ϕ maps external function names to their entry points, a propagation context $\mathcal{A}$ that dictates whether and how each external function propagates its argument labels to its return value label, and the call stack Ξ. Taint propagation policies returned by $\mathcal{A}$ are expressed as customizable mappings ρ from argument labels $\overline{\gamma}$ to return labels γ.

Operational Semantics. Figure 8 presents an operational semantics defining how taint labels propagate in an instrumented program. Expression judgments are large-step ($\Downarrow$), while command judgments are small-step ($\rightarrow_1$). At the IL level, expressions are pure and programs are non-reflective. Abstract machine configurations consist of tuples $\langle \sigma, \Delta, \lambda, \Xi, pc, \iota \rangle$, where *pc* is the program pointer and ι is the current instruction. Notation $\Delta[v \mapsto u]$ denotes function Δ with *v* remapped to *u*, and notation $\mathcal{P}[pc]$ refers to the program instruction at address *pc*. For brevity, we omit $\mathcal{P}$ from machine configurations, since it is fixed.

Rule VAL expresses the typical convention that hardcoded program constants are initially untainted ($\bot$). Binary operations are eager, and label their outputs with the join ($\sqcup$) of their operand labels. The semantics of $\texttt{load}(\tau, e)$ read the value stored in location *e*, where the label associated with the loaded value is obtained by propagation function ρ_{load}. Dually, $\texttt{store}(\tau, e_1, e_2)$ stores e_2 into location e_1, updating λ according to ρ_{store}. In C programs, these model pointer dereferences and dereferencing assignments, respectively. Parameterizing these rules in terms of abstract propagation functions ρ_{load} and ρ_{store} allows us to instantiate them with customized propagation policies at compile-time, as detailed in Section 3.2.

External function calls $\texttt{call}(\tau, f, \overline{e_1 \cdots e_n})$ evaluate arguments $\overline{e_1 \cdots e_n}$, create a new stack frame *fr*, and jump to the callee's entry point. Returns then consult propagation context $\mathcal{A}$ to appropriately label the value returned by the function based on the labels of its arguments. Context $\mathcal{A}$ can be customized by the user to specify how labels propagate through external libraries compiled without taint-tracking support.

Label Propagation Semantics. The operational semantics are parameterized by propagation functions ρ that can be instantiated to a specific propagation policy at compile-time. This provides a base framework through which we can study different propagation policies and their differing characteristics. Figure 9 presents three polymorphic[1] functions that can be used to instantiate propagation policies. On-load propagation policies instantiate ρ_{load}, while on-store policies instantiate ρ_{store}. The instantiations in Figure 9 define no-combine semantics (DFSan's on-store default), PCS (DFSan's on-load default), and our PC^{2}S extensions:

No-combine. The no-combine semantics (NCS) model a traditional, pointer-transparent propagation policy. Pointer labels are ignored during loads and stores, causing loaded and stored data retain their labels irrespective of the labels of the pointers being dereferenced.

Pointer-Combine Semantics. In contrast, PCS joins pointer labels with loaded and stored data labels during loads and stores. Using this policy, a value is tainted on-load (resp., on-store) if its source memory location (resp., source operand) is tainted or the pointer value dereferenced during the operation is tainted. If both are tainted with different labels, the labels are joined to obtain a new label that denotes the union of the originals.

Pointer Conditional-Combine Semantics. PC^{2}S generalizes PCS by conditioning the label-join on the static type of the data operand. If the loaded/stored data has pointer

[1] The functions are polymorphic in the sense that some of their arguments are types τ.

type, it applies the NCS rule; otherwise, it applies the PCS rule. The resulting label propagation for stores is depicted in Figure 10.

This can be leveraged to obtain the best of both worlds. PC^{2}S pointer taints retain most of the advantages of PCS—they can identify and track aliases to birthplaces of secrets, such as data structures where secrets are stored immediately after parsing, and they automatically propagate their labels to data stored there. But PC^{2}S resists PCS's over-tainting and label creep problems by avoiding propagation of pointer labels through levels of pointer indirection, which usually encode relationships with other data whose labels must remain distinct and separately managed.

Condition $(\tau \text{ is } ptr)$ in Figure 9 can be further generalized to any decidable proposition on static types τ. This feature is used to distinguish pointers that cross data ownership boundaries (e.g., pointers to other instances of the parent structure) from pointers that target value data (e.g., strings). The former receive NCS treatment by default to resist over-tainting, while the latter receive PCS treatment by default to capture secrets and keep the annotation burden low. In addition, PC^{2}S is at least as efficient as PCS because propagation policy ρ is partially evaluated at compile-time. Thus, the choice of NCS or PCS semantics for each pointer operation is decided purely statically, conditional upon the static types of the operands. The appropriate specialized propagation implementation is then in-lined into the resulting object code during compilation.

Example. To illustrate how each semantics propagate taint, consider the following IL pseudo-code, which revisits the linked-list example informally presented in §3.1.

```
1 store(id, request_id, get(s, id_size));
2 store(key, p[request_id]->key, get(s, key_size));
3 store(ctx_t*, p[request_id]->next, queue_head);
```

Input stream s includes a non-secret request identifier and a secret key of primitive type (e.g., unsigned long). If one labels stream s secret, then the public $request\_id$ becomes over-tainted in all three semantics, which is undesirable because a redaction of $request\_id$ may crash the program (when $request\_id$ is later used as an array index). A better solution is to label pointer p secret and employ PCS, which correctly labels the key at the moment it is stored. However, PCS additionally taints the $next$-pointer, leading to over-tainting of all the nodes in the containing linked-list, some of which may contain keys owned by other users. PC^{2}S avoids this over-tainting by exempting the next pointer from the combine-semantics. This preserves the data structure while correctly labeling the secret data it contains.

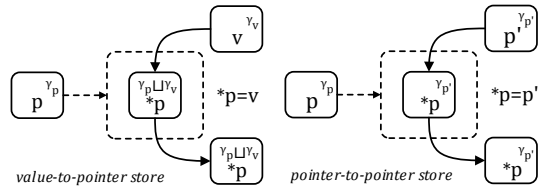


Fig. 10 PC^{2}S propagation policy on store commands.

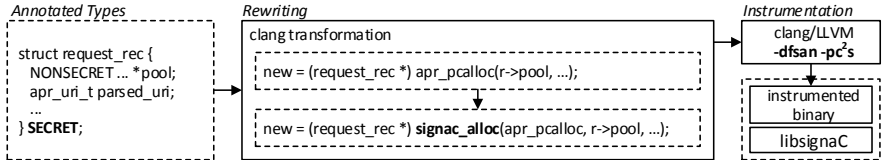


Fig. 11 Architectural overview of SIGNAC illustrating its three-step, static instrumentation process: (1) annotation of security-relevant types, (2) source-code rewriting, and (3) compilation with the sanitizer's instrumentation pass.

3.3 An Integrated Secret-redacting, Honey-patching Architecture

Figure 11 presents the architecture of SIGNAC[2] (Secret Information Graph iNstrumentation for Annotated C) [4], which leverages compiler-instrumented secret-redaction to achieve secret-sanitized process migration for secure honey-patching. At a high level, it consists of three components: (1) a source-to-source preprocessor, which (a) automatically propagates user-supplied, source-level type annotations to containing datatypes, and (b) in-lines taint introduction logic into dynamic memory allocation operations; (2) a modified LLVM compiler that instruments programs with PC^{2}S taint propagation logic during compilation; and (3) a runtime library that the instrumented code invokes during program execution to introduce taints and perform redaction.

Type attributes. Server code maintainers first annotate data structures containing secrets with the type qualifier `SECRET`. This instructs the taint-tracker to treat all instantiations (e.g., dynamic allocations) of these structures as taint sources. Additionally, qualifier `NONSECRET` may be applied to pointer fields within these structures to exempt them from PCS. The instrumentation pass generates NCS logic instead for operations involving such members. Finally, qualifier `SECRET_STR` may be applied to pointer fields whose destinations are dynamic-length byte sequences bounded by a null terminator (strings).

Type attribute rewriting. In the preprocessing step, the target application undergoes a source-to-source transformation pass that rewrites all dynamic allocations of annotated data types with taint-introducing wrappers. Implementing this transformation at the source level allows us to utilize the full type information that is available at the compiler's front-end, including purely syntactic attributes such as `SECRET` annotations. The implementation leverages Clang's tooling API [8] to traverse and apply the desired transformations directly into the program's AST.

Static Instrumentation. The instrumentation pass next introduces LLVM IR code during compilation that propagates taint labels during program execution. The implementation extends DFSan with the PC^{2}S label propagation policy specified in Section 3.2. Taint labels are represented as 16-bit integers, with new labels allocated sequentially

[2] named after *pointillism* co-founder Paul Signac

Listing 4 Taint-introducing memory allocations

```
1  #define signac_alloc(alloc, args...)  ({ \
2      void *__p = alloc ( args ); \
3      signac_taint(&__p, sizeof(void*)); \
4      __p; })
```

from a pool. DFSan maps (without reserving) the lower 32 TB of the process address space for *shadow memory*, which stores the taint labels of the values stored at the corresponding application memory addresses. At the front-end, compilation flags parametrize the label propagation policies for the store and load operations (*viz.*, NCS, PCS, or PC^{2}S).

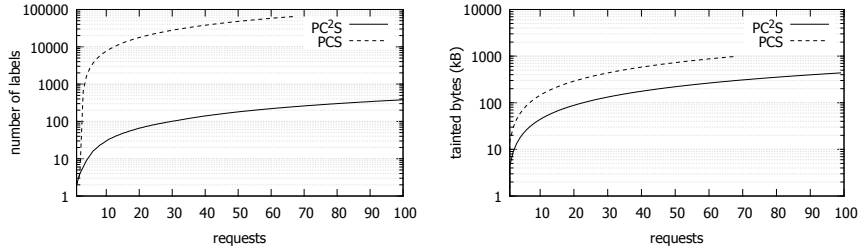
Runtime Library. The source-to-source rewriter and instrumentation phases in-line logic that calls a tiny dedicated library at runtime to introduce taints, handle special taint-propagation cases (e.g., string support), and check taints at sinks (e.g., during redaction). The library exposes three API functions:

- `signac_init(`pl`)`: initialize a tainting context with a fresh label instantiation pl for the current principal.
- `signac_taint(`$addr$`,`$size$`)`: taint each address in interval $[addr, addr{+}size)$ with pl.
- `signac_alloc(`$alloc$`,...)`: wrap allocator $alloc$ and taint the address of its returned pointer with pl.

Function `signac_init` instantiates a fresh taint label and stores it in a thread-global context, which function f of annotation `SECRET`$\langle f \rangle$ may consult to identify the owning principal at taint-introduction points. In typical web server architectures, this function is strategically hooked at the start of a new connection's processing cycle. Function `signac_taint` sets the labels of each address in interval $[addr, addr{+}size)$ with the label pl retrieved from the session's context.

Listing 4 details `signac_alloc`, which wraps allocations of `SECRET`-annotated data structures. This variadic macro takes a memory allocation function $alloc$ and its arguments, invokes it (line 2), and taints the address of the pointer returned by the allocator (line 3).

Example. To instrument a particular server application, such as Apache, SIGNAC requires two small, one-time developer interventions: First, add a call to `signac_init` at the start of a user session to initialize a new tainting context for the newly identified principal. Second, annotate the security-relevant data structures whose instances are to be tracked. For instance, in Apache, `signac_init` is called upon the acceptance of a new server connection, and annotated types include `request_rec`, `connection_rec`, `session_rec`, and `modssl_ctx_t`. These structures are where Apache stores URI parameters and request content information, private connection data such as remote IPs, key-value entries in user sessions, and encrypted connection information. The redaction scheme instruments the server with PC^{2}S. At redaction time, it scans the resulting shadow memory for labels denoting secrets owned by user sessions other than the attacker's, and redacts such secrets. The shadow memory and taint-tracking libraries are then unloaded, leaving a decoy process that masquerades as undefended and vulnerable.



(a) Label creeping behavior (b) Cumulative tally of bytes tainted

Fig. 12 Taint spread evaluation of PC^{2}S- and PCS-instrumented instances of the Apache web server.

Apache has been the most popular web server since April 1996 [3]. Its market share includes 54.5% of all active websites (the second, Nginx, has 11.97%) and 55.45% of the top-million websites (against Nginx with 15.91%) [26]. It is a robust, commercial-grade, feature-rich open-source software product comprised of 2.27M SLOC mostly in C [29], and has been tested on millions of web servers around the world. These characteristics make it a highly challenging, interesting, and practical flagship case study to test compiler-assisted taint-tracking and embedded honeypotting.

Taint spread evaluation. For comparing the taint spread properties of PC^{2}S and PCS, Apache's core modules for serving static and dynamic content, encrypting connections, and storing session data were annotated, omitting its optional modules. Altogether, approximately 45 type annotations need to be added to the web server source code, plus 30 SLOC to initialize the taint-tracker. Considering the size and complexity of Apache ($\sim$2.2M SLOC), PC^{2}S annotation burden is exceptionally light relative to prior taint-tracking approaches.

To test PC^{2}S's resistance to taint explosions, a stream of (non keep-alive) requests was submitted to each instrumented web server, recording a cumulative tally of distinct labels instantiated during taint-tracking. Figure 12a plots the results, comparing traditional PCS to PC^{2}S extensions. On Apache, traditional PCS is impractical, exceeding the maximum label limit in just 68 requests. In contrast, PC^{2}S instantiates vastly fewer labels (note that the y-axes are *logarithmic scale*). After extrapolation, an average 16,384 requests are required to exceed the label limit under PC^{2}S—well above the standard 10K-request TTL limit for worker threads.

Taint spread control is equally critical for preserving program functionality after redaction. To demonstrate, the experiment is repeated with a simulated intrusion after $n \in [1, 100]$ legitimate requests. Figure 12b plots the cumulative tally of how many bytes received a taint during the history of the run on Apache. In all cases, redaction crashed PCS-instrumented processes cloned after just 2–3 legitimate requests (due to erasure of over-tainted bytes). In contrast, PC^{2}S-instrumented processes never

Listing 5 Abbreviated patch for CVE-2014-6271

```
1 + if ((flags & SEVAL_FUNCDEF) && command->type != cm_function_def)
2 + {
3 +     internal_warning ("%s: ignoring function definition attempt", ...);
4 +     should_jump_to_top_level = 0;
5 +     last_result = last_command_exit_value = EX_BADUSAGE;
6 +     break;
7 + }
```

Listing 6 Honey-patch for CVE-2014-6271

```
1    if ((flags & SEVAL_FUNCDEF) && command->type != cm_function_def)
2    {
3 +    hp_fork();
4 +    hp_skip(
5        internal_warning ("%s: ignoring function definition attempt", ...);
6        should_jump_to_top_level = 0;
7        last_result = last_command_exit_value = EX_BADUSAGE;
8        break;
9 +    );
10   }
```

crashed during the experiment; their decoy clones continued running after redaction, impersonating vulnerable servers. This demonstrates the approach's facility to realize effective taint-tracking in legacy codes for which prior approaches fail.

4 Case Study: A Honey-Patch for Shellshock

The Shellshock GNU Bash remote command execution vulnerability (CVE-2014-6271) [28] was one of the most severe vulnerabilities in recent history, affecting millions of then-deployed web servers and other Internet-connected devices. This high impact combined with its ease of exploitation makes it a prime candidate for a practical application and evaluation of honey-patching.

Honey-patching Shellshock. Listing 5 shows an abbreviated, vendor-released patch in diff style for Shellshock. The patch introduces a conditional that validates environment variables passed to Bash, declining function definition attempts. Prior to this patch, attackers could take advantage of HTTP headers as well as other mechanisms to enable unauthorized access to the underlying system shell of remote targets. This patch exemplifies a common vulnerability mitigation: dangerous inputs or program states are detected via a boolean test, with positive detection eliciting a corrective action. The corrective action is typically readily distinguishable by attackers—in this case, a warning message is generated and the function definition is ignored.

Listing 6 presents an alternative, honey-patched implementation of the same patch. In response to a malicious input, the honey-patched application forks itself onto a confined, ephemeral, decoy environment, and behaves henceforth as an unpatched, vulnerable version of the software. Specifically, line 3 forks the user session to a decoy container, and macro `hp_skip` in line 4 elides the rejection in the decoy container so that the attack appears to have succeeded. Meanwhile, the attacker session in the original container

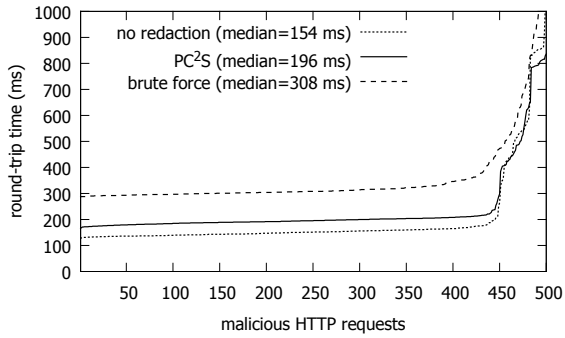


Fig. 13 Request round-trip times for attacker session forking on honey-patched Apache.

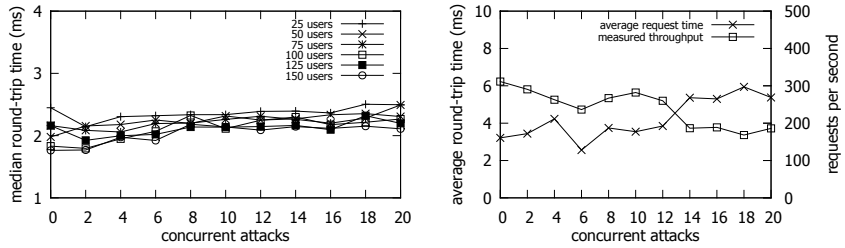
is safely terminated (having been forked to the decoy), and legitimate, concurrent connections continue unaffected.

As a result, adversaries attempting to exploit Shellshock in a victim server that has been honey-patched receive server responses that seem to indicate that the exploit has succeeded. However, the shell commands they inject are actually executing in a decoy environment stocked with disinformation for attackers to explore. Observe that the differences between the patch and the honey-patch are quite minor, except for the fixed cloning infrastructure that the honey-patch code references, and that can be maintained separately from the server code. This allows information technology and security administrators to easily reformulate patches into honey-patches after a new vulnerability disclosure is received, facilitating a quick, aggressive response to the threat. In general, only a superficial understanding of many patches is required to convert them to honey-patches of this form[3].

Decoy monitoring. Decoys host software monitors that allow defenders to collect rich and detailed fine-grained attack information. To minimize the performance impact on decoys, two powerful and highly efficient monitoring tools are implemented in REDHERRING [6]: *inotifywait* (to track modifications made to the file system), and *tcpdump* (to monitor ingress and egress of network packets). To avoid possible tampering with the collected data, all logs are stored outside the decoy environments. In addition, both monitoring tools are tuned to avoid generating spurious outputs (e.g., by excluding certain directories and limiting the monitored network traffic). As illustrative examples, attack data collected at decoys can be used to inform perimeter defenses to block exploit attempts against unpatched machines in the network, and to provide detailed threat intelligence to aid analysts in incidence response scenarios.

Performance Benchmarks. To evaluate honey-patching, one must determine the performance overhead imposed upon sessions forked to decoys (i.e., the impact on

[3] Araujo et al. [5] present a more systematic study of honey-patchable patches for all official security patches released for the Apache web server from 2005 to 2013. Overall, the analysis shows that roughly 65% of the patches analyzed are easily transformable into honey-patches.



(a) $0.6 \leq \sigma \leq 35,\ n = 10$ (b) $0.3 \leq \sigma \leq 1.2,\ 25 \leq \sigma \leq 94$

Fig. 14 Performance benchmarks. (a) Effect of concurrent attacks on legitimate HTTP request round-trip time on a single-node VM. (b) Stress test illustrating request throughput for a 3-node, load-balanced REDHERRING setup (workload $\approx$ 5K requests).

malicious users), and estimate the impact of honey-patching on the overall system performance (i.e., its impact on legitimate users). To obtain baseline measurements that are independent of networking overhead, the experiments in this section are executed locally on a single host using default Apache settings. Performance is measured in terms of HTTP request round-trip time.

Experimental setup. The target server was honey-patched against Shellshock and hosted a CGI shell script deployed atop Apache for processing user authentication in a web application created for this evaluation. All experiments were performed on a quad-core virtual machine (VM) with 8 GB RAM running 64-bit Ubuntu 14.04 (Trusty Tahr). Each LXC container running inside the VM was created using the official LXC Ubuntu template. We limited resource utilization on decoys so that a successful attack does not starve the host VM. The host machine is an Intel Xeon E5645 desktop running 64-bit Windows 7.

Session forking overhead. To evaluate the performance impact on attackers, benchmark results are reported for three honey-patched Apache deployments: (1) a baseline instance without memory redaction, (2) brute-force memory sweep redaction, and (3) our PC^{2}S redactor. Apache's server benchmarking tool (*ab*) is used to launch 500 malicious HTTP requests against each setup, each configured with a pool of 25 decoys. Figure 13 shows request round-trip times for each deployment. PC^{2}S redaction is about 1.6$\times$ faster than brute-force memory sweep redaction [5]; the former's request times average 0.196s, while the latter's average 0.308s. This significant reduction in cloning delay considerably improves the technique's deceptiveness, making it more transparent to attackers. To mask residual timing differences, the reverse proxy in a honey-patching architecture artificially delays all the non-forking responses to legitimate requests so that their round-trip times match those of the malicious requests that trigger the honey-patch.

Overall system overhead. To complete the evaluation, REDHERRING was also tested on a wide variety of workload profiles consisting of both legitimate users and attacker sessions on a single node. In this experiment, users and attackers trigger legitimate and

malicious HTTP requests, respectively. The request payload size is 2.4 KB, based on the median of KB per request measured by Google web metrics [17]. To simulate different usage profiles, the system is tested with 25–150 concurrent users, with 0–20 attackers. Figure 14a plots the results. Observe that for the various profiles analyzed, the HTTP request round-trip times remain approximately constant (ranging between 1.7 and 2.5 milliseconds) when increasing the number of concurrent malicious requests. This confirms that adding honey-patching capabilities has negligible performance impact on legitimate requests and users relative to traditional patches, even during concurrent attacks. This also shows that REDHERRING can cope with large workloads. This experiment assesses its baseline performance considering only one instance of the target server running on a single node virtual machine. In a real setting, several similar instances can be deployed using a web farm scheme to scale up to thousands of users, as presented next.

Stress testing. In this experiment, *ab* (Apache HTTP server benchmarking tool) is used to create a massive workload of legitimate users (more than 5,000 requests in 10 threads) for different attack profiles (0 to 20 concurrent attacks) against a three-node load-balanced setup of REDHERRING. Each VM is configured with a 2 GB RAM and one quad-core processor. The load balancer and the benchmark tool run on a separate VM on the same host machine. Apache runs with default settings (i.e., no fine tuning has been performed). As Figure 14b illustrates, the system can handle the strenuous workload imposed by the test suite. The average request time for legitimate users ranged from 2.5 to 5.9 milliseconds, with measured throughput ranging from 169 to 312 requests per second. In typical production settings this delay is amortized by the network latency (usually on the order of several tens of milliseconds). This result is important because it demonstrates that honey-patching can be realized for large-scale, performance critical software applications with minimal overheads for legitimate users.

5 Is Honey-Patching *Security through Obscurity*?

"Security through obscurity" (cf., [24]) has become a byword for security practices that rely upon an adversary's ignorance of the system design rather than any fundamental principle of security. History has demonstrated that such practices offer very weak security at best, and are dangerously misleading at worst, potentially offering an illusion of security that may encourage poor decision-making [1].

Security defenses based on deception potentially run the risk of falling into the "security through obscurity" trap. If the defense's deceptiveness hinges on attacker ignorance of the system design—details that defenders should conservatively assume will eventually become known by any suitably persistent threat actor—then any security offered by the defense might be illusory and therefore untrustworthy. It is therefore important to carefully examine the underlying basis upon which embedded honeypotting can be viewed as a security-enhancing technology.

Like all deception strategies, the effectiveness of honeypotting relies upon withholding certain secrets from adversaries (e.g., which software vulnerabilities have been honey-patched). But secret-keeping does not in itself disqualify honeypotting as obscurity-reliant. For example, modern cryptography is frequently championed as a hallmark of anti-obscurity defense despite its foundational assumption that adversaries lack knowledge of private keys, because disclosing the complete implementation details of crypto algorithms does not aid attackers in breaking cyphertexts derived from undisclosed keys.

Juels [18] defines *indistinguishability* and *secrecy* as two properties required for successful deployment of honey systems. These properties are formalized as follows:

> Consider a simple system in which $S = \{s_1, \ldots, s_n\}$ denotes a set of n objects of which one, $s^* = s_j$, for $j \in \{1, \ldots, n\}$ is the true object, while the other $n - 1$ are honey objects. The two properties then are:
>
> *Indistinguishability*: To deceive an attacker, honey objects must be hard to distinguish from real objects. They should, in other words, be drawn from a probability distribution over possible objects similar to that of real objects.
>
> *Secrecy*: In a system with honey objects, j is a secret. Honey objects can, of course, only deceive an attacker that doesn't know j, so j cannot reside alongside S. Kerckhoffs' principle therefore comes into play: the security of the system must reside in the secret, i.e., the distinction between honey objects and real ones, not in the mere fact of using honey objects.

Embedded honeypotting as a paradigm satisfies both these properties by design:

Indistinguishability derives from the inability of an attacker to determine whether an apparently successful attack is the result of exploiting an unpatched vulnerability or a honey-patch masquerading as an unpatched vulnerability. While absolute, universal indistinguishability is probably impossible to achieve, many forms of distinguishability can nevertheless be made arbitrarily difficult to discern. For example, honey-servers can exhibit response delay distributions that mimic those of unpatched servers to arbitrary degrees of precision (e.g., by artificially delaying legitimate, non-forking requests to match the distribution of malicious, forking requests, as described in Section 4).

Secrecy implies that the set of honey-patched vulnerabilities should be secret. However, full attacker knowledge of the design and implementation details of honey-patching does not disclose *which* vulnerabilities a defender has identified and honey-patched. Adapting Kerckhoffs' principle [20] for deception, a honey-patch is not detectable even if everything about the system, except the honey-patch, is public knowledge.

This argues that embedded honeypotting as a paradigm (and language-based software cyber deception in general) does not derive its security value from obscurity. Rather, its deceptions are based on well-defined secrets—specifically, the set of honey-patched vulnerabilities in target applications. Maintaining this confidentiality distinction between the publicness of honeypot design and implementation details, versus the secrecy of exactly which vulnerabilities instantiate those details, is important for crafting robust, effective deceptions.

6 Conclusion

This chapter introduced and formulated the concept of *embedded honeypots* as a language-level approach for arming production software with deceptive capabilities that mislead adversaries into wasting time and resources on phantom vulnerabilities and embedded decoy file systems. Embedded honeypots employ *honey-patches* to conceal from attackers the information of which software security vulnerabilities are patched, thereby degrading attackers' methods and disrupting their reconnaissance efforts.

To realize efficient, precise honey-patching of production web servers, a new statically-instrumented, dynamic taint analysis built upon the LLVM compiler infrastructure is highlighted. The implementation significantly improves the feasibility of dynamic taint-tracking for low-level legacy code that stores secrets in graph data structures. To ease the programmer's annotation burden and avoid taint explosions suffered by prior approaches, it introduces a novel pointer-combine semantics that resists taint over-propagation through graph edges. Deceptive servers self-redact their address spaces in response to intrusions, affording defenders a new tool for attacker monitoring and disinformation.

Embedded honeypots differ from traditional honeypots in that they reside within the actual, mission-critical software systems that attackers are seeking to penetrate, and not as independent decoy systems. Thus, embedded honeypots offer advanced deceptive remediations against informed adversaries who can identify and avoid traditional honeypots. In order to be adoptable, embedded honeypots imbue production server software with deceptive capabilities without impairing its performance or intended functionality. These new capabilities make cyber attacks significantly more costly and risky for their perpetrators, and give defenders more time and opportunity to detect and thwart incoming attacks.

References

1. ANDERSON, R. Why information security is hard – an economic perspective. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)* (2001), pp. 358–365.
2. ANSEL, J., ARYA, K., AND COOPERMAN, G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2009), pp. 1–12.
3. APACHE. Apache HTTP server project. `http://httpd.apache.org`, 2014.
4. ARAUJO, F., AND HAMLEN, K. W. Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception. In *Proc. 24th USENIX Security Symp.* (2015).
5. ARAUJO, F., HAMLEN, K. W., BIEDERMANN, S., AND KATZENBEISSER, S. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)* (2014), pp. 942–953.
6. ARAUJO, F., SHAPOURI, M., PANDEY, S., AND HAMLEN, K. Experiences with honey-patching in active cyber security education. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)* (2015).
7. BRINGER, M. L., CHELMECKI, C. A., AND FUJINOKI, H. A survey: Recent advances and future trends in honeypot research. *International Journal of Computer Network and Information Security 4*, 10 (2012).

8. CLANG. clang.llvm.org. http://clang.llvm.org.
9. CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI)* (2005), vol. 2, pp. 273–286.
10. CORBET, J. TCP Connection Repair. http://lwn.net/Articles/495304, 2012.
11. CRIU. Checkpoint/Restore In Userspace. http://criu.org, 2014.
12. DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Tainting is not pointless. *ACM/SIGOPS Operating Systems Review (OSR) 44*, 2 (2010), 88–92.
13. DFSAN. Clang DataFlowSanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html.
14. DUELL, J. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Tech. Rep. LBNL-54941, U. California at Berkeley, 2002.
15. GEROFI, B., FUJITA, H., AND ISHIKAWA, Y. An efficient process live migration mechanism for load balanced distributed virtual environments. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)* (2010), pp. 197–206.
16. GOOGLE. Protocol Buffers. https://code.google.com/p/protobuf, 2014.
17. GOOGLE. Web metrics. https://developers.google.com/speed/articles/web-metrics, 2014.
18. JUELS, A. A bodyguard of lies: the use of honey objects in information security. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies* (2014), ACM, pp. 1–4.
19. KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)* (2011).
20. KERCKHOFFS, A. La cryptographie militaire. *Journal Sciences Militaires IX* (1883), 5–38.
21. LATTNER, C., AND ADVE, V. S. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)* (2004), pp. 75–88.
22. LIGHTTPD. Lighttpd server project. http://www.lighttpd.net, 2014.
23. LXC. Linux containers. http://linuxcontainers.org, 2014.
24. MERKOW, M. S., AND BREITHAUPT, J. *Information Security: Principles and Practices*. Pearson Education, 2014.
25. MILOJIČIĆ, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process migration. *ACM Computing Surveys 32*, 3 (2000), 241–299.
26. NETCRAFT. Are there really lots of vulnerable Apache web servers? http://news.netcraft.com/archives/2014/02/07, 2014.
27. NGINX. Nginx server project. http://nginx.org, 2014.
28. NIST. The Shellshock Bash Vulnerability. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271, Sep. 2014.
29. OHLOH. Apache HTTP server statistics. http://www.ohloh.net/p/apache, 2014.
30. PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proceedings of the Conference on USENIX Annual Technical Conference (ATEC)* (1999), pp. 15–15.
31. PINGREE, L. Emerging Technology Analysis: Deception Techniques and Technologies Create Security Technology Business Opportunities. *Gartner, Inc.* (July 2015). ID:G00278434.
32. PROVOS, N. A virtual honeypot framework. In *USENIX Security Symposium* (2004), vol. 173.
33. SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)* (2010), pp. 317–331.
34. SLOWINSKA, A., AND BOS, H. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)* (2009), pp. 61–74.
35. SPITZNER, L. *Honeypots: Tracking Hackers*. Addison-Wesley Longman, 2002.
36. SPITZNER, L. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, 2 (2003), 15–23.

37. THONNARD, O., AND DACIER, M. A framework for attack patterns' discovery in honeynet data. *Digital Investigation 5, Supplement* (2008), S128 – S139. The Proceedings of the Eighth Annual {DFRWS} Conference.
38. VORIS, J., JERMYN, J., BOGGS, N., AND STOLFO, S. Fox in the trap: thwarting masqueraders via automated decoy document deployment. In *Proceedings of the 8thEuropean Workshop on System Security* (2015), ACM, p. 3.
39. WANG, C., MUELLER, F., ENGELMANN, C., AND SCOTT, S. L. Proactive process-level live migration in HPC environments. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (2008).
40. WHITAKER, A., COX, R. S., SHAW, M., AND GRIBBLE, S. D. Constructing services with interposable virtual hardware. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)* (2004), pp. 169–182.
41. YUILL, J., ZAPPE, M., DENNING, D., AND FEER, F. Honeyfiles: Deceptive files for intrusion detection. In *Proceedings of the 5thIEEE Int. Workshop on Information Assurance* (2004), IEEE, pp. 116–122.