Compiler-instrumented, Dynamic Secret-Redaction of Legacy Processes for Attacker Deception

Frederico Araujo and Kevin W. Hamlen
The University of Texas at Dallas
{*frederico.araujo, hamlen*}@*utdallas.edu*

Abstract

An enhanced dynamic taint-tracking semantics is presented and implemented, facilitating fast and precise runtime secret redaction from legacy processes, such as those compiled from C/C++. The enhanced semantics reduce the annotation burden imposed upon developers seeking to add secret-redaction capabilities to legacy code, while curtailing over-tainting and label creep.

An implementation for LLVM's DataFlow Sanitizer automatically instruments taint-tracking and secret-redaction support into annotated C/C++ programs at compile-time, yielding programs that can self-censor their address spaces in response to emerging cyber-attacks. The technology is applied to produce the first information flow-based honey-patching architecture for the Apache web server. Rather than merely blocking intrusions, the modified server deceptively diverts attacker connections to secret-sanitized process clones that monitor attacker activities and disinform adversaries with honey-data.

1 Introduction

Redaction of sensitive information from documents has been used since ancient times as a means of concealing and removing secrets from texts intended for public release. As early as the 13th century B.C., Pharaoh Horemheb, in an effort to conceal the acts of his predecessors from future generations, so thoroughly located and erased their names from all monument inscriptions that their identities weren't rediscovered until the 19th century A.D. [22]. In the modern era of digitally manipulated data, *dynamic taint analysis* (cf., [40]) has become an important tool for automatically tracking the flow of secrets (*tainted data*) through computer programs as they execute. Taint analysis has myriad applications, including program vulnerability detection [5, 6, 9, 25, 33, 34, 37, 45, 46], malware analysis [19, 20, 36, 48], test set generation [3, 42], and information leak detection [4, 14, 21, 23, 24, 49].

Our research introduces and examines the associated challenge of secret redaction from program process images. Safe, efficient redaction of secrets from program address spaces has numerous potential 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 honey-potting.

A recent instantiation of the latter is *honey-patching* [2], which proposes crafting software security patches in such a way that future attempted exploits of the patched vulnerabilities appear successful to attackers. This frustrates attacker vulnerability probing, and affords defenders opportunities to disinform attackers by divulging "fake" secrets in response to attempted intrusions. In order for such deceptions to succeed, honey-patched programs must be imbued with the ability to impersonate unpatched software with all secrets replaced by honey-data. That is, they require a technology for rapidly and thoroughly redacting all secrets from the victim program's address space at runtime, yielding a vulnerable process that the attacker may further penetrate without risk of secret disclosure.

Realizing such runtime process secret redaction in practice educes at least two significant research 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. 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.

Our approach builds upon the LLVM compiler's [31] DataFlow Sanatizer (DFSan) infrastructure [18], 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, we augment DFSan with a declarative, type annotation-based secret-labeling mechanism for easier secret identification; and we introduce a new label propagation semantics, called *Pointer Conditional-Combine Semantics* (PC^2S), that 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 redacting secret process data whilst preserving process operation after redaction.

Our innovations are showcased through the development of a taint tracking-based honey-patching framework for three production web servers, including the popular Apache HTTP server (~2.2M SLOC). The modified servers respond to detected intrusions by transparently forking attacker sessions to unpatched process clones in confined decoy environments. Runtime redaction preserves attacker session data without preserving data owned by other users, yielding a deceptive process that continues servicing the attacker without divulging secrets. The decoy can then monitor attacker strategies, harvest attack data, and disinform the attacker with honey-data in the form of false files or process data.

Our contributions can be summarized as follows:

- We introduce a pointer tainting methodology through which secret sources are derived from statically annotated data structures, lifting the burden of identifying classification code-points in legacy C code.
- We propose and formalize taint propagation semantics that accurately track secrets while controlling taint spread. Our solution is implemented as a small extension to LLVM, allowing it to be applied to a large class of COTS applications.
- We implement a memory redactor for secure honey-patching. Evaluation shows that our implementation is both more efficient and more secure than previous pattern-matching based redaction approaches.
- Implementations and evaluations for three production web servers demonstrate that the approach is feasible for large-scale, performance-critical software with reasonable overheads.

Listing 1: 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  }
```

2 Approach Overview

We first outline practical limitations of traditional dynamic taint-tracking for analyzing dataflows in server applications, motivating our research. We then overview our approach and its application to the problem of redacting secrets from runtime process memory images.

2.1 Dynamic Taint Analysis

Dynamic taint analyses enforce *taint policies*, which specify how data confidentiality and integrity classifications (*taints*) are introduced, propagated, and checked as a program executes. *Taint introduction* rules specify taint sources—typically a subset of program inputs. *Taint propagation* rules define how taints flow. For example, the result of summing tainted values might be a sum labeled with the union (or more generally, the *lattice join*) of the taints of the summands. *Taint checking* is the process of reading taints associated with data, usually to enforce an information security policy. Taints are usually checked at data usage or disclosure points, called *sinks*.

Extending 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 1 exemplifies the problem using a code excerpt from the Apache web server [1]. 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 over-taints the user name (and the colon delimiter, and the rest of the stream), leading to excessive label creep—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.

2.2 Sourcing & Tracking Secrets

To ease this burden, we 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 1 as secret, users of our system may add type qualifier `SECRET_STR` to the password field's declaration in its abstract datatype definition. Our modified LLVM compiler responds to this static annotation by dynamically tainting 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 $\texttt{SECRET}\langle f\rangle$ identifies a user-implemented 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, we leverage and extend DFSan's *pointer-combine semantics (PCS)* feature, which optionally combines (i.e., joins) 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 simple 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 [17, 27, 40, 43].

Listing 2: Abbreviated Apache's session record struct

```
typedef struct {
    NONSECRET apr_pool_t *pool;
    NONSECRET apr_uuid_t *uuid;
    SECRET_STR const char *remote_user;
    apr_table_t *entries;
    ...
} SECRET session_rec;
```

To address this shortcoming, we introduce a new, generalized PC^{2}S semantics that augments PCS 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. We find 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, our 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). Our work 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.

Users of our system 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 2 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. Our solution thus complements existing approaches.

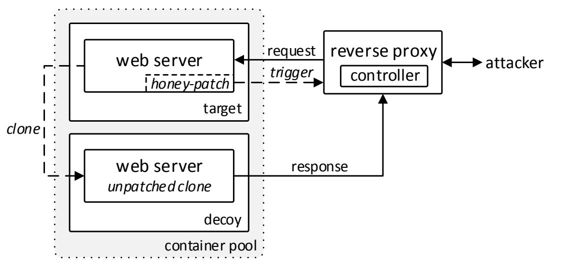


Figure 1: Architectural overview of honey-patching.

2.3 Application Study: Honey-Patching

Our discoveries are applied to realize practical, efficient honey-patching of legacy web servers for attacker deception. Typical software security patches fix newly discovered vulnerabilities at the price of advertising to attackers which systems have been patched. Cyber-criminals therefore easily probe today's Internet for vulnerable software, allowing them to focus their attacks on susceptible targets.

Honey-patching, depicted in Figure 1, is a recent strategy for frustrating such attacks. In response to malicious inputs, honey-patched applications clone the attacker session onto a confined, ephemeral, decoy environment, which behaves henceforth as an unpatched, vulnerable version of the software. This potentially augments the server with an embedded honeypot that waylays, monitors, and disinforms criminals.

Highly efficient cloning is critical for such architectures, since response delays risk alerting attackers to the deception. The cloning process must therefore rapidly locate and redact all secrets from the process address space, yielding a runnable process with only the attacker's session data preserved. Moreover, redaction must not be overly conservative. If redaction crashes the clone with high probability, or redacts obvious non-secrets, this too alerts the attacker. To our knowledge, no prior taint-tracking approach satisfies all of these demanding performance, precision, and legacy-maintainability requirements. We therefore select honey-patching of Apache as our flagship case-study.

3 Formal Semantics

For explanatory precision, we formally define our new taint-tracking semantics in terms of the simple, typed intermediate language (IL) in Figure 2, inspired by prior work [40]. The simplified IL abstracts irrelevant details of LLVM's IR language, capturing only those features needed to formalize our analysis.

3.1 Language Syntax

Programs $\mathcal{P}$ are lists of commands, denoted $\overline{c}$. Commands consist of variable assignments, pointer-dereferencing as-

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

Figure 2: Intermediate representation syntax.

signments (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 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.

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), we include a function table ϕ that 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.

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

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1\rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2\rangle}{\sigma, \Delta, \lambda \vdash \Diamond_b(\tau,\ e_1,\ e_2) \Downarrow \langle u_1 \Diamond_b u_2, \gamma_1 \sqcup \gamma_2\rangle}\ \textsc{BinOp} \qquad \frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma\rangle}{\sigma, \Delta, \lambda \vdash \texttt{load}(\tau,\ e) \Downarrow \langle \sigma(u), \rho_{load}(\tau, \gamma, \lambda(u))\rangle}\ \textsc{Load}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma\rangle \quad \Delta' = \Delta[v \mapsto u] \quad \lambda' = \lambda[v \mapsto \gamma]}{\langle \sigma, \Delta, \lambda, \Xi, pc, v := e\rangle \rightarrow_1 \langle \sigma, \Delta', \lambda', \Xi, pc+1, \mathcal{P}[pc+1]\rangle}\ \textsc{Assign}$$

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1\rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2\rangle \quad \sigma' = \sigma[u_1 \mapsto u_2] \quad \lambda' = \lambda[u_1 \mapsto \rho_{store}(\tau, \gamma_1, \gamma_2)]}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{store}(\tau, e_1, e_2)\rangle \rightarrow_1 \langle \sigma', \Delta, \lambda', \Xi, pc+1, \mathcal{P}[pc+1]\rangle}\ \textsc{Store}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma\rangle \qquad \sigma, \Delta, \lambda \vdash e_{(u\,?\,1\,:\,0)} \Downarrow \langle u', \gamma'\rangle}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{br}(e, e_1, e_0)\rangle \rightarrow_1 \langle \sigma, \Delta, \lambda, \Xi, u', \mathcal{P}[u']\rangle}\ \textsc{Cond}$$

$$\frac{\begin{array}{c}\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1\rangle \quad \cdots \quad \sigma, \Delta, \lambda \vdash e_n \Downarrow \langle u_n, \gamma_n\rangle \\ \Delta' = \Delta[\overline{params_f} \mapsto \overline{u_1 \cdots u_n}] \quad \lambda' = \lambda[\overline{params_f} \mapsto \overline{\gamma_1 \cdots \gamma_n}] \quad fr = \langle f, pc+1, \Delta, \overline{\gamma_1 \cdots \gamma_n}\rangle\end{array}}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{call}(\tau, f, \overline{e_1 \cdots e_n})\rangle \rightarrow_1 \langle \sigma, \Delta', \lambda', fr :: \Xi, \phi(f), \mathcal{P}[\phi(f)]\rangle}\ \textsc{Call}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma\rangle \quad fr = \langle f, pc', \Delta', \overline{\gamma}\rangle \quad \lambda' = \lambda[v_{ret} \mapsto \mathcal{A}\, f\, \overline{\gamma}]}{\langle \sigma, \Delta, \lambda, fr :: \Xi, pc, \texttt{ret}(\tau, e)\rangle \rightarrow_1 \langle \sigma, \Delta'[v_{ret} \mapsto u], \lambda', \Xi, pc', \mathcal{P}[pc']\rangle}\ \textsc{Ret}$$

Figure 3: Operational semantics of a generalized label propagation semantics.

3.2 Operational Semantics

Figure 3 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 hard-coded 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 §3.3.

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.

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	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := (\tau \text{ is } ptr)\ ?\ \gamma_2 : (\gamma_1 \sqcup \gamma_2)$

Figure 4: Polymorphic functions modeling no-combine, pointer-combine, and PC^{2}S label propagation policies.

3.3 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 4 presents three polymorphic 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 4 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.

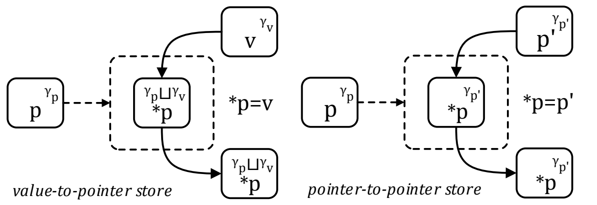


Figure 5: PC^{2}S propagation policy on store commands.

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 type, it applies the NCS rule; otherwise, it applies the PCS rule. The resulting label propagation for stores is depicted in Figure 5.

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 (τ is ptr) in Figure 4 can be further generalized to any decidable proposition on static types τ. We use this feature 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 IL pseudo-code in Listing 3, which revisits the linked-list example informally presented in §2.2. 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.

Listing 3: IL pseudo-code for storing public ids and secret keys from an unstructured input stream into a linked list.

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

4 Implementation

Figure 6 presents an architectural overview of our implementation, SignaC[1] (Secret Information Graph iNstrumentation for Annotated C). At a high level, the implementation 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. Each component is described below.

4.1 Source-Code Rewriting

Type attributes. Users 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).

To avoid augmenting the source language's grammar, these type qualifiers are defined using source-level attributes (specified with `__attribute__`) followed by a specifier. `SECRET` uses the `annotate` specifier, which defines a purely syntactic qualifier visible only at the compiler's front-end. In contrast, `NONSECRET` and `SECRET_STR` are required during the back-end instrumentation. To this end, we leverage Quala [39], which extends LLVM with an overlay type system. Quala's `type_annotate` specifier propagates the type qualifiers throughout the IL code.

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

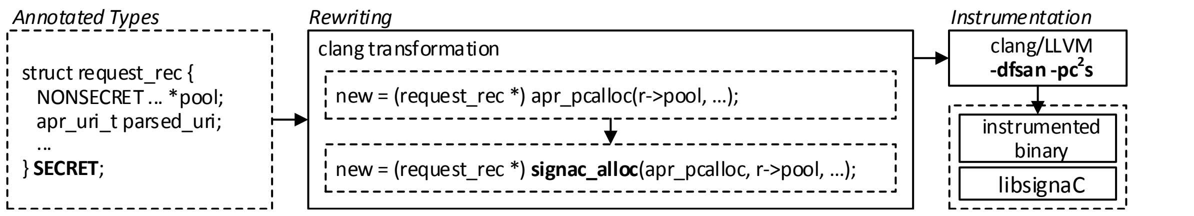


Figure 6: Architectural overview of SignaC illustrating its three-step, static instrumentation process: (1) annotation of security-relevant types, (2) source-code rewriting, and (3) compilation with the sanitizer's instrumentation pass.

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.

Our implementation leverages Clang's tooling API [12] to traverse and apply the desired transformations directly into the program's AST. At a high-level, the rewriting algorithm takes the following steps:

1. It first amasses a list of all *security-relevant datatypes*, which are defined as (a) all structs and unions annotated `SECRET`, (b) all types defined as aliases (e.g., via typedef) of security-relevant datatypes, and (c) all structs and unions containing secret-relevant datatypes not separated from the containing structure by a level of pointer indirection (e.g., nested struct definitions). This definition is recursive, so the list is computed iteratively from the transitive closure of the graph of datatype definition references.
2. It next finds all calls to memory allocation functions (e.g., malloc, calloc) whose return values are *explicitly* or *implicitly* cast to a security-relevant datatype. Such calls are wrapped in calls to SignaC's runtime library, which dynamically introduces an appropriate taint label to the newly allocated structure.

The task of identifying memory allocation functions is facilitated by a user-supplied list that specifies the memory allocation API. This allows the rewriter to handle programs that employ custom memory management. For example, Apache defines custom allocators in its Apache Portable Runtime (APR) memory management interface.

4.2 PC^{2}S Instrumentation

The instrumentation pass next introduces LLVM IR code during compilation that propagates taint labels during program execution. Our implementation extends DFSan with the PC^{2}S label propagation policy specified in §3.

Taint representation. To support a large number of taint labels, DFSan adopts a low-overhead representation of labels as 16-bit integers, with new labels allocated sequentially from a pool. Rather than reserving 2^n labels to represent the full power set of a set of n primitive taints, DFSan lazily reserves labels denoting non-singleton sets on-demand. When a label union operation is requested at a join point (e.g., during binary operations on tainted operands), the instrumentation first checks whether a new label is required. If a label denoting the union has already been reserved, or if one operand label subsumes the other, DFSan returns the already-reserved label; otherwise, it reserves a fresh union label from the label pool. The fresh label is defined by pointers to the two labels that were joined to form it. Union labels are thus organized as a dynamically growing binary DAG—the *union table*.

This strategy benefits applications whose label-joins are sparse, visiting only a small subset of the universe of possible labels. Our PC^{2}S semantics' curtailment of label creep thus synergizes with DFSan's lazy label allocation strategy, allowing us to realize taint-tracking for legacy code that otherwise exceeds the maximum label limit. This benefit is further evidenced in our evaluation (§5).

Table 1 shows the memory layout of an instrumented program. 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. This layout allows for efficient lookup of shadow addresses by masking and shifting the application's addresses. Labels of values not stored in memory (e.g., those stored in machine registers or optimized away at compile-time) are tracked at the IL level in SSA registers, and compiled to suitable taint-tracking object code.

Function calls. Propagation context $\mathcal{A}$ defined in §3 models label propagation across external library function calls, expressed in DFSan as an Application Binary Interface (ABI). The ABI lists functions whose label-propagation

Table 1: Memory layout of an instrumented program.

Start	End	Memory Region
0x700000008000	0x800000000000	application memory
0x200000000000	0x200200000000	union table
0x000000010000	0x200000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

behavior (if any) should be replaced with a fixed, user-defined propagation policy at call sites. For each such function, the ABI specifies how the labels of its arguments relate to the label of its return value.

DFSan natively supports three such semantics: (1) *discard*, which corresponds to propagation function $\rho_{dis}(\overline{\gamma}) := \bot$ (return value is unlabeled); (2) *functional*, corresponding to propagation function $\rho_{fun}(\overline{\gamma}) := \bigsqcup \overline{\gamma}$ (label of return value is the union of labels of the function arguments); and (3) *custom*, denoting a custom-defined label propagation wrapper function.

DFSan pre-defines an ABI list that covers glibc's interface. Users may extend this with the API functions of external libraries for which source code is not available or cannot be instrumented. For example, to instrument Apache with `mod_ssl`, we mapped OpenSSL's API functions to the ABI list. In addition, we extended the custom ABI wrappers of *memory transfer functions* (e.g., `strcpy`, `strdup`) and *input functions* (e.g., `read`, `pread`) to implement PC^{2}S. For instance, we modified the wrapper for `strcpy(`*dest*`,` *src*`)` to taint $dest$ with $\gamma_{src} \sqcup \gamma_{dest}$ when instrumenting code under PC^{2}S.

Static instrumentation. The instrumentation pass is placed at the end of LLVM's optimization pipeline. This ensures that only memory accesses surviving all compiler optimizations are instrumented, and that instrumentation takes place just before target code is generated. Like other LLVM *transform* passes, the program transformation operates on LLVM IR, traversing the entire program to insert label propagation code. At the front-end, compilation flags parametrize the label propagation policies for the store and load operations discussed in §3.3.

String handling. Strings in C are not first-class types; they are implemented as character pointers. C's type system does not track their lengths or enforce proper termination. This means that purely static typing information is insufficient for the instrumentation to reliably identify strings or propagate their taints to all constituent bytes on store. To overcome this problem, users must annotate secret-containing, string fields with `SECRET_STR`. This cues the runtime library to taint up to and including the pointee's null terminator when a string is assigned to such a field. For safety, our runtime library (see §4.3) zeros the first byte of all fresh memory allocations, so that uninitialized strings are always null-terminated.

Listing 4: Store instruction instrumentation

```
1      Value* Shadow = DFSF.getShadow(SI.getValueOperand());
2   +  if (Cl_PC2S_OnStore) {
3   +    Type *t = SI.getValueOperand()->getType();
4   +    if (!t->isPointerTy() || !isExemptPtr(&SI)) {
5   +      Value *PtrShadow = DFSF.getShadow(SI.getPointerOperand());
6   +      Shadow = DFSF.combineShadows(Shadow, PtrShadow, &SI);
7   +    }
8   +  }
9      ...
10     DFSF.storeShadow(SI.getPointerOperand(), Size, Align, Shadow, &SI);
11  +  if (Cl_PC2S_OnStore) {
12  +    if (isSecretStr(&SI)) {
13  +      Value *Str = IRB.CreateBitCast(v, Type::getInt8PtrTy(Ctx));
14  +      IRB.CreateCall2(DFSF.DFS.DFSanSetLabelStrFn, Shadow, Str);
15  +    }
16  +  }
```

Listing 5: Load instruction instrumentation

```
1     Value *Shadow = DFSF.loadShadow(LI.getPointerOperand(), Size, ...);
2   + if (Cl_PC2S_OnLoad) {
3   +   if (!isExemptPtr(&LI)) {
4   +     Value *PtrShadow = DFSF.getShadow(LI.getPointerOperand());
5   +     Shadow = DFSF.combineShadows(Shadow, PtrShadow, &LI);
6   +   }
7   + }
8     ...
9     DFSF.setShadow(&LI, Shadow);
```

Store instructions. Listing 4 summarizes the instrumentation procedure for stores in diff style. By default, DFSan instruments NCS on store instructions: it reads the shadow memory of the value operand (line 1) and copies it onto the shadow of the pointer operand (line 10). If PC^{2}S is enabled (lines 2 and 11), the instrumentation consults the static type of the value operand and checks whether it is a non-pointer or non-exempt pointer field (which also subsumes `SECRET_STR`) in lines 3–4. If so, the shadows of the pointer and value operands are joined (lines 5–6), and the resulting label is stored into the shadow of the pointer operand. If the instruction stores a string annotated with `SECRET_STR`, the instrumentation calls a runtime library function that copies the computed shadow to all bytes of the null-terminated string (lines 12–15).

Load instructions. Listing 5 summarizes the analogous instrumentation for load instructions. First, the instrumentation loads the shadow of the value pointed by the pointer operand (line 1). If PC^{2}S is enabled (line 2), then the instrumentation checks whether the dereferenced pointer is tainted (line 3). If so, the shadow of the pointer operand is joined with the shadow of its value (lines 4–5), and the resulting label is saved to the shadow (line 9).

Memory transfer intrinsics. LLVM defines intrinsics for standard memory transfer operations, such as `memcpy` and `memmove`. These functions accept a source pointer src, a destination pointer dst, and the number of bytes len to be transferred. DFSan's default instrumentation destructively copies the shadow associated with src to the shadow of dst, which is not the intended propagation policy of PC^{2}S. We therefore instrument these functions as shown in Listing 6. The instrumentation reads the shadows of src and dst (lines 2–3), computes the union of the two shadows (line 4), and stores the combined shadows to the shadow of dst (line 5).

4.3 Runtime Library

Runtime support for the type annotation mechanism is encapsulated in a tiny C library, allowing for low coupling

Listing 6: Memory transfer intrinsics instrumentation

```
+ if (CI_PC2S_OnStore && !isExemptPtr(&I)) {
+   Value *DestShadow = DFSF.getShadow(I.getDest());
+   Value *SrcShadow = DFSF.getShadow(I.getSource());
+   DestShadow = DFSF.combineShadows(SrcShadow, DestShadow, &I);
+   DFSF.storeShadow(I.getDest(), Size, Align, DestShadow, &I);
+ }
```

Listing 7: Taint-introducing memory allocations

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

between a target application and the sanitizer's logic. The source-to-source rewriter and instrumentation phases inline logic that calls this 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 7 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).

4.4 Apache Instrumentation

To instrument a particular server application, such as Apache, our approach 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.

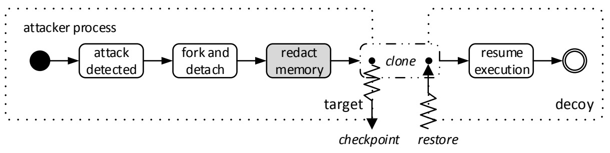


Figure 7: Honey-patch response to an intrusion attempt.

5 Evaluation

This section demonstrates the practical advantages and feasibility of our approach for retrofitting large legacy C codes with taint-tracking, through the development and evaluation of a honey-patching memory redaction architecture for three production web servers. All experiments were performed on a quad-core VM with 8 GB RAM running 64-bit Ubuntu 14.04. The host machine is an Intel Xeon E5645 workstation running 64-bit Windows 7.

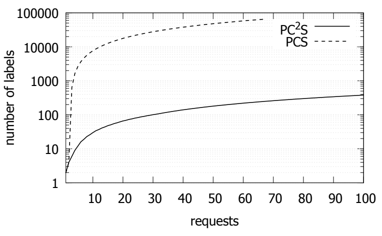
5.1 Honey-patching

Figure 7 illustrates how honey-patches respond to intrusions by cloning attacker sessions to decoys. Upon intrusion detection, the honey-patch forks a shallow, local clone of the victim process. The cloning step redacts all secrets from the clone's address space, optionally replacing them with honey-data. It then resumes execution in the decoy by emulating an unpatched implementation. This impersonates a successful intrusion, luring the attacker away from vulnerable victims, and offering defenders opportunities to monitor and disinform adversaries.

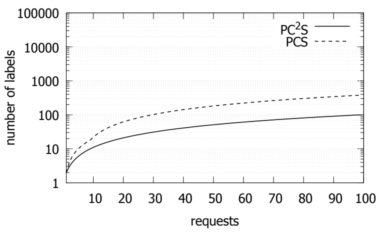
Prior honey-patches implement secret redaction as a brute-force memory sweep that identifies and replaces plaintext string secrets. This is both slow and unsafe; the sweep constitutes a majority of the response delay overhead during cloning [2], and it can miss binary data secrets difficult to express reliably as regular expressions. Using SignaC, we implemented an information flow-based redaction strategy for honey-patching that is faster and more reliable than prior approaches.

Our redaction scheme instruments the server with dynamic taint-tracking. 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.

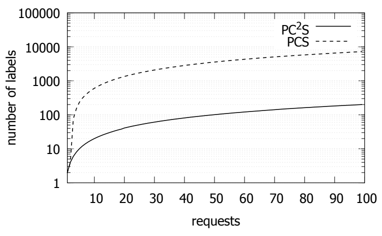
Evaluated software. We implemented taint tracking-based honey-patching for three production web servers: Apache, Nginx, and Lighttpd. Apache and Nginx are the top two servers of all active websites, with 50.1% and 14.8% market share, respectively [32]. Apache comprises 2.27M SLOC mostly in C [35]. Nginx and Lighttpd are smaller, having about 146K and 138K SLOC, respectively. All three are commercial-grade, feature-rich, open-source



(a) Apache



(b) Nginx



(c) Lighttpd

Figure 8: Experiment comparing label creeping behavior of PC^{2}S and PCS on Apache, Nginx, and Lighttpd.

software products without any built-in support for information flow tracking.

To augment these products with PC^{2}S-style taint-tracking support, we manually annotated secret-storing structures and pointer fields. Altogether, we added approximately 45, 30, and 25 such annotations to Apache, Nginx, and Lighttpd, respectively. For consistent evaluation comparisons, we only annotated Apache's core modules for serving static and dynamic content, encrypting connections, and storing session data; we omitted its optional modules. We also manually added about 20–30 SLOC to each server to initialize the taint-tracker. Considering the sizes and complexity of these products, we consider the PC^{2}S annotation burden exceptionally light relative to prior approaches.

5.2 Taint Spread

Over-tainting protection. To test our approach's resistance to taint explosions, we submitted a stream of (non keep-alive) requests to each instrumented web server, recording a cumulative tally of distinct labels instantiated during taint-tracking. Figure 8 plots the results, comparing traditional PCS to our 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, we conclude that 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, we repeated the experiment with a simulated intrusion after $n \in [1, 100]$ legitimate requests. Figure 9 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 crashed; their decoy clones continued running after redaction, impersonating vulnerable servers. This demonstrates our

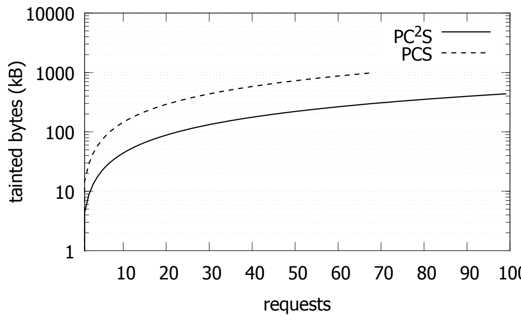


Figure 9: Cumulative tally of bytes tainted on Apache.

Table 2: Honey-patched security vulnerabilities

Software	Version	CVE-ID	Description
Bash[1]	4.3	CVE-2014-6271	Improper parsing of environment variables
OpenSSL[1]	1.0.1f	CVE-2014-0160	Buffer over-read in heartbeat protocol extension
Apache	2.2.21	CVE-2011-3368	Improper URL validation
Apache	2.2.9	CVE-2010-2791	Improper timeouts of keep-alive connections
Apache	2.2.15	CVE-2010-1452	Bad request handling
Apache	2.2.11	CVE-2009-1890	Request content length out of bounds
Apache	2.0.55	CVE-2005-3357	Bad SSL protocol check

[1]tested with Apache 2.4.6

approach's facility to realize effective taint-tracking in legacy codes for which prior approaches fail.

Under-tainting protection. To double-check that PC^{2}S redaction was actually erasing all secrets, we created a workload of legitimate post requests with pre-seeded secrets to a web-form application. We then automated exploits of the honey-patched vulnerabilities listed in Table 2, including the famous Shellshock and Heartbleed vulnerabilities. For each exploit, we ran the legacy, brute-force memory sweep redactor after SignaC's redactor to confirm that the former finds no secrets missed by the latter. We also manually inspected memory dumps of each clone to confirm that none of the pre-seeded secrets

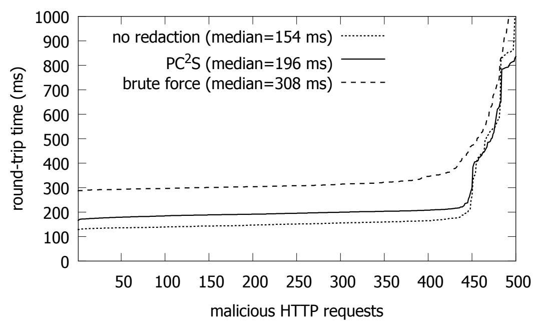


Figure 10: Request round-trip times for attacker session forking on honey-patched Apache.

survived. In all cases, the honey-patch responds to the exploits as a vulnerable decoy server devoid of secrets.

5.3 Performance

Redaction performance. To evaluate the performance overhead of redacting secrets, we benchmarked 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. We used Apache's server benchmarking tool (*ab*) to launch 500 malicious HTTP requests against each setup, each configured with a pool of 25 decoys.

Figure 10 shows request round-trip times for each deployment. PC^{2}S redaction is about $1.6\times$ faster than brute-force memory sweep redaction; 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. Nginx and Lighttpd also exhibit improved response times of 16% (0.165s down to 0.138s) and 21% (0.155s down to 0.122s), respectively.

Taint-tracking performance. To evaluate the performance overhead of the static instrumentation, three Apache setups were tested: a static-content HTML website ($\sim$20 KB page size), a CGI-based Bash application that returns the server's environment variables, and a dynamic PHP website displaying the server's configuration. For each web server setup, *ab* was executed with four concurrency levels c (i.e., the number of parallel threads). Each run comprises 500 concurrent requests, plotted in ascendant order of their round-trip times (RTT).

Figure 11 shows the results for $c = 1, 10, 50$, and 100, and the average overheads observed for each test profile are summarized in Table 3. Our measurements show overheads of $2.4\times$, $1.1\times$, and $0.3\times$ for the static-content, CGI, and PHP websites, respectively, which is consistent with dynamic taint-tracking overheads reported in the prior literature [41]. Since server computation accounts for only about 10% of overall web site response delay in

Table 3: Average overhead of instrumentation

Benchmark	$c = 1$	$c = 10$	$c = 50$	$c = 100$
Static	2.50	2.34	2.56	2.32
CGI Bash	1.29	0.98	1.00	0.97
PHP	0.41	0.37	0.30	0.31

practice [44], this corresponds to observable overheads of about 24%, 11%, and 3% (respectively).

While such overhead characterizes feasibility, it is irrelevant to deception because unpatched, patched, and honey-patched vulnerabilities are all slowed equally by the taint-tracking instrumentation. The overhead therefore does not reveal which apparent vulnerabilities in a given server instance are genuine patching lapses and which are deceptions, and it does not distinguish honey-patched servers from servers that are slowed by any number of other factors (e.g., fewer computational resources). In addition, it is encouraging that high relative overheads were observed primarily for static websites that perform little or no significant computation. This suggests that the more modest 3% overhead for computationally heavier PHP sites is more representative of servers for which computational performance is an issue.

6 Discussion

6.1 Approach Limitations

Our research significantly eases the task of tracking secrets within standard, pointer-linked, graph data-structures as they are typically implemented in low-level languages, like C/C++. However, there are many non-standard, low-level programming paradigms that our approach does not fully support automatically. Such limitations are discussed below.

Pointer Pre-aliases. PC^{2}S fully tracks all pointer aliases via taint propagation starting from the point of taint-introduction (e.g., the code point where a secret is first assigned to an annotated structure field after parsing). However, if the taint-introduction policy misidentifies secret sources too late in the program flow, dynamic tracking cannot track pointer *pre-aliases*—aliases that predate the taint-introduction. For example, if a program first initializes string p_1, then aliases $p_2 := p_1$, and finally initializes secret-annotated field f via $f := p_1$, PC^{2}S automatically labels p_1 (and f) but not pre-alias p_2.

In most cases this mislabeling of pre-aliases can be mitigated by enabling PC^{2}S both on-load and on-store. This causes secrets stored via p_2 to receive the correct label on-load when they are later read via p_1 or f. Likewise, secrets read via p_2 retain the correct label if they were earlier stored via p_1 or f. Thus, only data stored *and* read purely using independent pre-alias p_2 remain untainted.

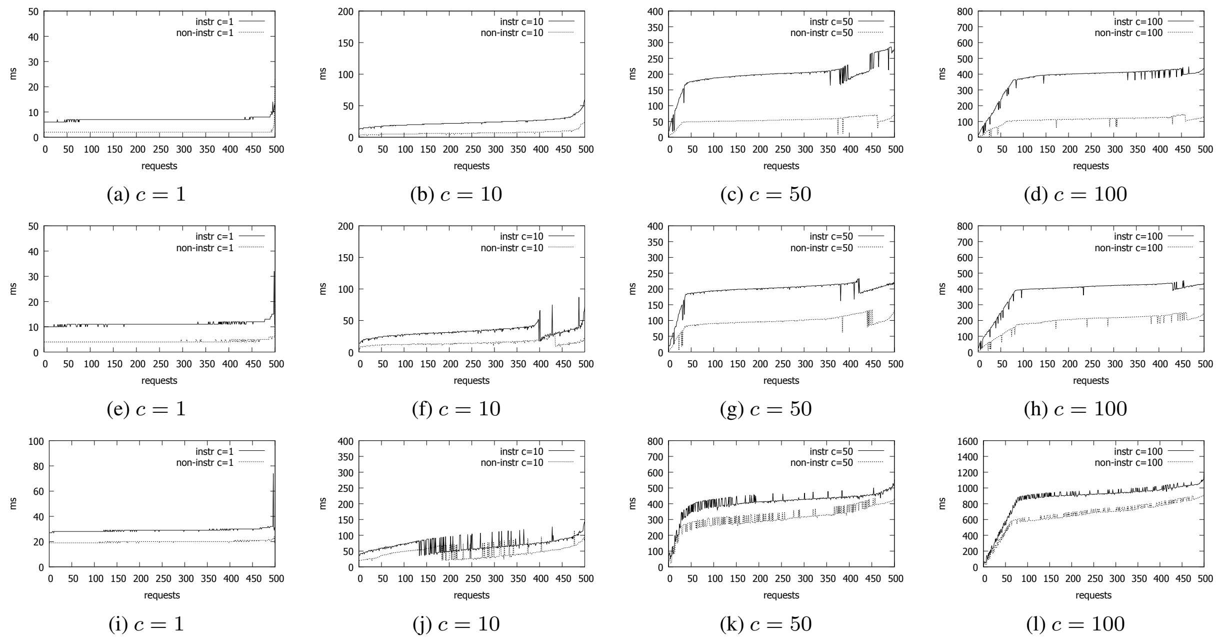


Figure 11: Dynamic taint-tracking performance (measured in request round-trip times) with varying concurrency c for a static web site (a–d), Bash CGI application (e–h), and PHP application (i–l).

This is a correct enforcement of the user's policy, since the policy identifies $f := p_1$ as the taint source, not p_2. If this treatment is not desired, the user must therefore specify a more precise policy that identifies the earlier origin of p_1 as the true taint source (e.g., by manually inserting a dynamic classification operation where p_1 is born), rather than identifying f as the taint source.

Structure granularity. Our automation of taint-tracking for graph data-structures implemented in low-level languages leads to taint annotations at the granularity of whole struct declarations, not individual value fields. Thus, all non-pointer fields within a secret-annotated C struct receive a common taint under our semantics. This coarse granularity is appropriate for C programs since such programs can (and often do) refer to multiple data fields within a given struct instance using a common pointer. For example, *marshalling* is typically implemented as a pointer-walk that reads a byte stream directly into all data fields (but not the pointer fields) of a struct instance byte-by-byte. All data fields therefore receive a common label after marshalling.

Reliable support for structs containing secrets of mixed taint therefore requires a finer-grained taint-introduction policy than is expressible by declarative annotations of C structure definitions. Such policies must be operationally specified in C through runtime classifications at secret-introducing code points. Our focus in this research is on automating the much more common case where each node of the graph structure holds secrets of uniform classification, toward lifting the user's annotation burden for this most common case.

Dynamic-length secrets. Our implementation provides built-in support for a particularly common form of dynamic-length secret—null-terminated strings. This can be extended to support other forms of dynamic-length secrets as needed. For example, strings with an explicit length count rather than a terminator, fat and bounded pointers [26], and other variable-length, dynamically allocated, data structures can be supported through the addition of an appropriate annotation type and a dynamic taint-propagating function that extends pointer taints to the entire pointee during assignments.

Implicit Flows. Our dynamic taint-tracking tracks explicit information flows, but not implicit flows that disclose information through control-flows rather than dataflows. Tracking implicit flows generally requires static information flow analysis to reason about disclosures through inaction (non-observed control-flows) rather than merely actions. Such analysis is often intractable (and generally undecidable) for low-level languages like C, whose control-flows include unstructured and dynamically computed transitions.

Likewise, dynamic taint-tracking does not monitor side-channels, such as resource consumption (e.g., memory or power consumption), runtimes, or program termination, which can also divulge information. For our problem domain (program process redaction), such channels are largely irrelevant, since attackers may only exfiltrate information after redaction, which leaves no secrets for the attacker to glean, directly or indirectly.

6.2 Process Memory Redaction

Our research introduces live process memory image sanitization as a new problem domain for information flow analysis. Process memory redaction raises unique challenges relative to prior information flow applications. It is exceptionally sensitive to over-tainting and label creep, since it must preserve process execution (e.g., for process debugging, continued service availability, or attacker deception); it demands exceptionally high performance; and its security applications prominently involve large, low-level, legacy codes, which are the most frequent victims of cyber-attacks. Future work should expand the search for solutions to this difficult problem to consider the suitability of other information flow technologies, such as static type-based analyses.

6.3 Language Compatibility

While our implementation targets one particularly ubiquitous source language (C/C++), our general approach is applicable to other similarly low-level languages, as well as scripting languages whose interpreters are implemented in C (e.g., PHP, Bash). Such languages are common choices for implementing web services, and targeting them is therefore a natural next step for the web security thrust of our research.

7 Related Work

Dynamic tracking of in-memory secrets. Dynamic taint-tracking lends itself as a natural technique for tracking secrets in software. It has been applied to study sensitive data lifetime (i.e., propagation and duration in memory) in commodity applications [10, 11], analyze spyware behavior [19, 48], and impede the propagation of secrets to unauthorized sinks [21, 23, 49].

TaintBochs [10] uses whole-system simulation to understand secret propagation patterns in several large, widely deployed applications, including Apache, and implements *secure deallocation* [11] to reduce the risk of exposure of in-memory secrets. Panorama [48] builds a system-level information-flow graph using process emulation to identify malicious software tampering with information that was not intended for their consumption. Egele et al. [19] also utilize whole-system dynamic tainting to analyze spyware behavior in web browser components. While valuable, the performance impact of whole-system analyses—often on the order of 2000% [10, 19, 48]—remains a significant obstacle, rendering such approaches impractical for most live, high-performance, production server applications.

More recently, there has been growing interest in runtime detection of information leaks [21, 49]. For instance, TaintDroid [21] extends Android's virtualized architecture with taint-tracking support to detect misuses of users' private information across mobile apps. TaintEraser [49] uses dynamic instrumentation to apply taint analysis on binaries for the purpose of identifying and blocking information leaking to restricted output channels. To achieve this, it monitors and rewrites sensitive bytes escaping to the network and the local file system. Our work adopts a different strategy to instrument secret-redaction support into programs, resulting in applications that can proactively respond to attacks by self-censoring their address spaces with minimal overhead.

Pointer taintedness. In security contexts, many categories of widely exploited, memory-overwrite vulnerabilities (e.g., format string, memory corruption, buffer overflow) have been recognized as detectable by dynamic taint-checking on pointer dereferences [7, 8, 15, 16, 28]. Hookfinder [47] employs data and pointer tainting semantics in a full-system emulation approach to identify malware hooking behaviors in victim systems. Other systems follow a similar technique to capture system-wide information-flow and detect privacy-breaching malware [19, 48].

With this high practical utility come numerous theoretical and practical challenges for effective pointer tainting [17, 27, 43]. On the theoretical side, there are varied views of how to interpret a pointer's label. (Does it express a property of the pointer value, the values it points to, values read or stored by dereferencing the pointer, or all three?) Different taint tracking application contexts solicit differing interpretations, and the differing interpretations lead to differing taint-tracking methodologies. Our contributions include a pointer tainting methodology that is conducive to tracking in-memory secrets.

On the practical side, imprudent pointer tainting often leads to taint explosion in the form of over-tainting or label-creep [40, 43]. This can impair the feasibility of the analysis and increase the likelihood of crashes in programs that implement data-rewriting policies [49]. To help overcome this, sophisticated strategies involving pointer injection (PI) analysis have been proposed [16, 28]. PI uses a taint bit to track the flow of legitimate pointers and another bit to track the flow of untrusted data, disallowing dereferences of tainted values that do not have a corresponding pointer tainted. Our approach uses static typing information in lieu of PI bits to achieve lower runtime overheads and broader compatibility with low-level legacy code.

Application-level instrumentation. Much of the prior work on dynamic taint analysis has employed dynamic binary instrumentation (DBI) frameworks [9,13,29,33,38, 49] to enforce taint-tracking policies on software. These approaches do not require application recompilation, nor do they depend on source code information.

However, despite many optimization advances over the years, dynamic instrumentation still suffers from significant performance overheads, and therefore cannot support high-performance applications, such as the redaction speeds required for attacker-deceiving honey-patching of production server code. Our work benefits from research advances on static-instrumented, dynamic data flow analysis [6, 18, 30, 46] to achieve both high performance and high accuracy by leveraging LLVM's compilation infrastructure to instrument taint-propagating code into server code binaries.

8 Conclusion

PC^{2}S 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. Our LLVM implementation extends C/C++ with declarative type qualifiers for secrets, and instruments programs with taint-tracking capabilities at compile-time.

The new infrastructure is applied to realize efficient, precise honey-patching of production web servers for attacker deception. The deceptive servers self-redact their address spaces in response to intrusions, affording defenders a new tool for attacker monitoring and disinformation.

9 Acknowledgments

The research reported herein was supported in part by AFOSR Award FA9550-14-1-0173, NSF CAREER Award #1054629, and ONR Award N00014-14-1-0030. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of the AFOSR, NSF, or ONR.

References

[1] APACHE. Apache HTTP server project. http://httpd.apache.org, 2014.

[2] ARAUJO, F., HAMLEN, K. W., BIEDERMANN, S., AND KATZENBEISSER, S. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2014), pp. 942–953.

[3] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)* (2010), pp. 1–11.

[4] BAUER, L., CAI, S., JIA, L., PASSARO, T., STROUCKEN, M., AND TIAN, Y. Run-time monitoring and formal analysis of information flows in Chromium. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2015).

[5] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: The world's fastest taint tracker. In *Proc. Int. Sym. Recent Advances in Intrusion Detection (RAID)* (2011), pp. 1–20.

[6] CHANG, W., STREIFF, B., AND LIN, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2008), pp. 39–50.

[7] CHEN, S., PATTABIRAMAN, K., KALBARCZYK, Z., AND IYER, R. K. Formal reasoning of various categories of widely exploited security vulnerabilities by pointer taintedness semantics. In *Proc. IFIP TC11 Int. Conf. Information Security (SEC)* (2004), pp. 83–100.

[8] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., AND IYER, R. K. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. Int. Conf. Dependable Systems and Networks (DSN)* (2005), pp. 378–387.

[9] CHENG, W., ZHAO, Q., YU, B., AND HIROSHIGE, S. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. IEEE Sym. Computers and Communications (ISCC)* (2006), pp. 749–754.

[10] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole system simulation. In *Proc. USENIX Security Symposium* (2004), pp. 321–336.

[11] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. USENIX Security Symposium* (2005), pp. 331–346.

[12] CLANG. clang.llvm.org. http://clang.llvm.org.

[13] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A generic dynamic taint analysis framework. In *Proc. ACM/SIGSOFT Int. Sym. Software Testing and Analysis (ISSTA)* (2007), pp. 196–206.

[14] COX, L. P., GILBERT, P., LAWLER, G., PISTOL, V., RAZEEN, A., WU, B., AND CHEEMALAPATI, S. Spandex: Secure password tracking for Android. In *Proc. USENIX Security Sym.* (2014).

[15] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Raksha: A flexible information flow architecture for software security. In *Proc. Int. Sym. Computer Architecture (ISCA)* (2007), pp. 482–493.

[16] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proc. USENIX Security Symposium* (2008), pp. 395–410.

[17] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Tainting is not pointless. *ACM/SIGOPS Operating Systems Review (OSR) 44*, 2 (2010), 88–92.

[18] DFSAN. Clang DataFlowSanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html.

[19] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic spyware analysis. In *Proc. USENIX Annual Technical Conf. (ATC)* (2007), pp. 233–246.

[20] EGELE, M., SCHOLTE, T., KIRDA, E., AND KRUEGEL, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR) 44*, 2 (2012), 1–42.

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

[22] EPIGRAPHIC SURVEY, THE ORIENTAL INSTITUTE OF THE UNIVERSITY OF CHICAGO, Ed. *Reliefs and Inscriptions at Luxor Temple*, vol. 1–2 of *The University of Chicago Oriental Institute*

Publications. Oriental Institute of the University of Chicago, Chicago, 1994, 1998.

[23] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. Int. Conf. Trust and Trustworthy Computing (TRUST)* (2012), pp. 291–307.

[24] Gu, A. B., Li, X., Li, G., Champion, Chen, Z., Qin, F., and Xuan, D. D2Taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources. In *Proc. IEEE Conf. Computer Communications (INFOCOM)* (2013), pp. 791–799.

[25] Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. Practical taint-based protection using demand emulation. In *Proc. ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys)* (2006), pp. 29–41.

[26] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conf. (ATC)* (2002), pp. 275–288.

[27] Kang, M. G., McCamant, S., Poosankam, P., and Song, D. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2011).

[28] Katsunuma, S., Kurita, H., Shioya, R., Shimizu, K., Irie, H., Goshima, M., and Sakai, S. Base address recognition with data flow tracking for injection attack detection. In *Proc. Pacific Rim Int. Sym. Dependable Computing (PRDC)* (2006), pp. 165–172.

[29] Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proc. Conf. Virtual Execution Environments (VEE)* (2012), pp. 121–132.

[30] Lam, L. C., and Chiueh, T. A general dynamic information flow tracking framework for security applications. In *Proc. Annual Computer Security Applications Conf. (ACSAC)* (2006), pp. 463–472.

[31] Lattner, C., and Adve, V. S. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. IEEE/ACM Int. Sym. Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)* (2004), pp. 75–88.

[32] Netcraft. Web surver survey. http://news.netcraft.com/archives/category/web-server-survey, January 2015.

[33] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2005).

[34] Nguyen-tuong, A., Guarnieri, S., Greene, D., and Evans, D. Automatically hardening web applications using precise tainting. In *Proc. IFIP TC11 Int. Conf. Information Security (SEC)* (2005), pp. 372–382.

[35] Ohloh. Apache HTTP server statistics. http://www.ohloh.net/p/apache.

[36] Papagiannis, I., Migliavacca, M., and Pietzuch, P. PHP Aspis: Using partial taint tracking to protect against injection attacks. In *Proc. USENIX Conf. Web Application Development (WebApps)* (2011).

[37] Portokalidis, G., Slowinska, A., and Bos, H. Argos: An emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys)* (2006), pp. 15–27.

[38] Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., and Wu, Y. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. Int. Sym. Microarchitecture (MICRO)* (2006), pp. 135–148.

[39] Sampson, A. Quala: Type qualifiers for LLVM/Clang. https://github.com/sampsyo/quala, 2014.

[40] 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 *Proc. IEEE Sym. Security & Privacy (S&P)* (2010), pp. 317–331.

[41] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A fast address sanity checker. In *Proc. USENIX Annual Technical Conf. (ATC)* (2012), pp. 309–318.

[42] Sezer, E. C., Ning, P., Kil, C., and Xu, J. Memsherlock: An automated debugger for unknown memory corruption vulnerabilities. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2007), pp. 562–572.

[43] Slowinska, A., and Bos, H. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proc. ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys)* (2009), pp. 61–74.

[44] Souders, S. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly, 2007.

[45] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004), pp. 85–96.

[46] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. USENIX Security Symposium* (2006).

[47] Yin, H., Liang, Z., and Song, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2008).

[48] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2007), pp. 116–127.

[49] Zhu, D. Y., Jung, J., Song, D., Kohno, T., and Wetherall, D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review (OSR) 45*, 1 (2011), 142–154.