V-DIFT: Vector-Based Dynamic Information Flow Tracking with Application to Locating Cryptographic Keys for Reverse Engineering

Antonio M. Espinoza,
Jeffrey Knockel,
and Jedidiah R. Crandall
University of New Mexico
{amajest,jeffk,crandall}@cs.unm.edu

Pedro Comesaña-Alfaro
University of Vigo
pcomesan@gts.uvigo.es

***Abstract*—Dynamic Information Flow Tracking (DIFT) is a technique for tracking information as it flows through a program's execution. DIFT systems track information by tainting data and propagating the taint marks throughout execution. These systems are designed to have minimal overhead and thus often miss indirect flows. If indirect flows were propagated naively overtainting would result, whereas propagating them effectively causes overhead. We describe the design and evaluation of a system intended for offline analysis, such as reverse engineering, that can track information through indirect flows. Our system, V-DIFT, uses a vector of floating point values for each taint mark. The use of vectors enables us to track a taint's provenance and handle indirect flows, trading off some performance for these abilities. These indirect flows *via* control and address dependencies are thought to be critical to tracking information flow of cryptographic programs. Therefore we tested V-DIFT's effectiveness by automatically locating keys in simple programs that use a variety of symmetric cryptographic algorithms found in three common libraries. This application does not require that the program run in real time, just that it be much faster than a manual approach. Our V-DIFT implementation tests average 3.6 seconds, and with the right parameters can identify memory locations that contain keys for 24 out of 27 algorithms tested. Our results show that many cryptographic algorithm implementations' address and/or control dependencies must be tracked for DIFT to be effective.**

I. Introduction

Modern Dynamic Information Flow Tracking (DIFT) (often called Dynamic Taint Analysis or DTA) systems (see, *e.g.*, [20], [6], [17], [4], or [12]) are designed to be run on production systems. Thus, they were created with performance in mind with the focus being on simple applications such as tracking malicious inputs from the network into control data or tracking how data is copied throughout a system.

In this paper we consider a different design point: applications where propagating taint through indirect flows is necessary and where relatively high performance overhead is tolerable. We consider the following problem: Given a program binary and an input, analyze a *single* trace of the program and identify any cryptographic keys that are used anywhere in the trace. Analysis of a single trace is important, because in reverse engineering it is often not realistic to expect the analyst to provide the tool with a representative set of traces. Our V-DIFT implementation can accomplish this based entirely on information flow analysis, without any prior knowledge or heuristics about the cryptographic algorithm itself.

Our overarching goal is not to replace modern DIFT systems, but to apply DIFT to new areas of research that can benefit from it through offline analysis. However to accommodate these new applications, tradeoffs must be made, namely, performance for the ability to track indirect information flows.

A. Adequate Information

What modern DIFT systems possess in speed they lack in information necessary to meet our desired applications. Being only able to taint a location with a tag (many of which are binary) is very limiting and can cause problems with information flow tracking; the worst being the problem of overtainting. Overtainting usually occurs when the instruction pointer (EIP in x86) becomes tainted, which leads to the program's entire memory to quickly becoming tainted. Thus no useful information can be derived from the DIFT system. Our focus in this paper is on measuring the actual information flow in a meaningful way. We accomplish this by using vectors as taint marks.

Another problem that stems from a lack of information in DIFT systems is taint attribution, a problem that is common in systems with binary taint marks. Being able to attribute the taint of a memory location to its source is essential to the task of locating cryptographic keys in memory. These problems exist because of current systems' lack of ability to handle indirect flows in a general way. Indirect flows can take the form of what Suh *et al.* [20] define as address and control dependencies.

In V-DIFT we use vectors as a representation for taint tags, as well as provide a way to combine them so that they are much more useful in telling us about the information flow of a program trace. Vectors allow us the ability to easily combine taint and propagate information about its source. Using vectors allows the use of simple linear algebra to approximate quantitative information flow, so that we are never faced with the dilemma of whether or not to propagate taint. Propagating taint is simply a matter of degree, with taint being combined *via* vector addition, and comparing taint values can be accomplished by simply taking the cosine similarity. This allows us to answer important questions of attribution as well as use the amplitude of taint marks as a proxy for how much mutual information there is between a taint sink and any taint source.

V-DIFT's use of vectors does not give a complete picture of the information flow but an approximation. This is due to the fact that we are only approximating information flow instead of strictly adhering to any information-theoretic definition of quantitative information flow. This relaxation of information flow allows us to "tease out" useful information from the trace without over- or undertainting rendering the results useless, thus supplying adequate information for analysis.

B. Terminology

The vocabulary used when discussing DIFT until now has largely been the same as static information flow described by Denning [9]. In this paper, we propose separate terminology for static systems *vs.* dynamic systems, because the goals of each are different and the information available is not equivalent. We view *explicit* and *implicit* flows as being properties of a program and thus useful when describing static information flow analyses. Similarly, we use *direct* and *indirect* for describing the information flow analysis of program runs or traces. This makes *implicit* and *explicit* flows properties of programs, and *direct* and *indirect* flows properties of traces. Since DIFT can be viewed as a single-pass analysis of a program trace, *direct* and *indirect* flows are the terminology we prefer in this paper.

We use the term *indirect* because some *implicit* flows cannot be measured in a DIFT system that does not execute both paths of the code. For example for a trace through this code:

```
y = 1;
if (x == 0) {
  y = 2;
}
```

. . . we can never know that information flowed from x to y when x is non-zero because the code `y = 2` was never executed. This is an implicit flow. By contrast, an indirect flow only occurs when x is equal to 0. In the case where x is non-zero the assignment $y = 2$ is never observed and there is no flow, except an implicit flow which DIFT can never reason about because it operates on single traces. We define an *indirect* flow as occurring when information dependent on program input determines from where and to where information flows, so a DIFT system can only measure this flow of information from x to y when the indirect flow actually occurs. Thus, more traces of the program with different inputs yield different information flows.

By contrast, *direct* flows are flows that will occur on any execution of an instruction regardless of the program's input. For example in: `add eax ebx` it is clear that information always flows from `ebx` to `eax` (note that in x86 the result of the addition is stored in `eax`). By this definition copy and computation dependencies are direct flows.

C. Contributions

This paper makes the following contributions:

- We propose V-DIFT, a general and practical solution to the problem of overtainting that uses vectors as tags and approximates quantitative information flow for offline analysis.
- We provide insights about the overtainting problem. Specifically, indirect flows result from a lack of necessary information when a DIFT system must make a propagation decision. Separating this problem from the more general problem of implicit flows suggests that indirect flows can be solved and need not be a fundamental limitation of DIFT systems.
- We demonstrate a DIFT-based method for locating the source of cryptographic keys based entirely on information flow, *i.e.*, without relying on any information or heuristics about the cryptographic algorithm employed or making any assumptions about how it is implemented.

II. Related Work

All work on DIFT systems either assumes that the information flow will be reasoned about statically or uses simple heuristics that only work in narrow domains and do not generalize. To the best of our knowledge, our work is the first to handle indirect flows in a general way.

TaintBochs [3] is the first paper that we know of to apply DIFT in the modern sense. TaintBochs was used to analyze data lifetime for privacy reasons. Shortly after, several research groups concurrently and independently developed DIFT as a way to track malicious inputs and prevent attacks [20], [6], [17], [5]. These early DIFT systems largely ignored indirect flows, or had propagation rules based on very simple heuristics. For example, in Suh *et al.* [20] address dependencies are not propagated if the address is calculated using a scaled index base (an x86 construct for calculating addresses). In addition, control dependencies were not propagated at all in Suh *et al.*.

In Minos [6], [7] many assumptions were made, such as address dependencies only being propagated for 8- and 16-bit loads and stores, but not for 32-bit loads and stores (Minos was based on a 32-bit system). For a detailed discussion see Crandall *et al.* [7], Slowinska and Bos [19], and King *et al.* [13].

Later research in DIFT systems attempted to address indirect flows. For example, many DIFT system frameworks that are designed for flexibility [8], [18], [21], [4] enable address and/or control dependencies to be tracked, but offer no solution to the overtainting problem. Panorama [24] relies on the user to manually label for which address and control dependencies the DIFT system should propagate tags.

DTA++ [12] only taints indirect flows deemed *culprit implicit flows* found through static analysis. This requires multiple traces retrieved through special test executions that fully exercise the portion of code that potentially has undertainting when executed ignoring control dependencies. DTA++ also suffers from a lack of byte-level taint attribution.

V-DIFT offers a way to handle indirect flows, that is not only general, but can be applied to a single trace (important for reverse engineering applications) and is byte-level attributable.

The main distinguishing features between existing analyses of cryptography in binary programs and our work are: (1) past work focuses on locating and identifying the cryptographic algorithm while our work focuses on locating the key; and, (2) past work relies on both the input/output relationship of cryptography algorithms and detailed knowledge about specific cryptography algorithms (e.g. [2], [25], and [14]), while our work simply examines the information flow of a trace and requires no algorithm-specific information about the symmetric cipher. V-DIFT allows us to define symmetric algorithms in general terms based on the key. Specifically, we can look for a small set of contiguous bytes that affect a significant amount of the output in a way that is correlated. Further analysis to detect which cryptographic algorithm is being used once the key is located is relatively straightforward.

Gröbert *et al.* [10] finds cryptographic primitives in binaries and identifies the algorithm. Their approach is relatively general compared to other past works, but relies on heuristics about instruction mixes, sequences, and loops. Because their technique requires no algorithm-specific signatures, templates, high-level reference implementations, and their experimental methodology is based on readily-available, open source libraries, we based our experimental methodology on that of Gröbert *et al.*. Wang *et al.* [22] and Caballero *et al.* [1] were earlier works that are similar works to Gröbert *et al.*. More recently, Hosfelt [11] presents an approach that is also similar to Gröbert *et al.* but is based on machine learning.

Whelan *et al.* [23] propose a method to characterize cryptography using their PIRATE system, which is a DIFT system. However they only illustrate the correlation between inputs and outputs of cryptographic algorithms. PIRATE required the tracking of address dependencies to be turned on to track AES CBC mode correctly. Lutz [15] presents an approach to analyze cryptography that is based on DTA [17], suffering from its limitations, and the DIFT (*i.e.*, taint) analysis is only a small part of Lutz's technique. Ming *et al.* [16] also present cryptography algorithm detection as one application of DIFT, in a manner that is similar to Whelan *et al.*.

III. V-DIFT IMPLEMENTATION

We use vectors as taint marks to store and utilize information unavailable to other DIFT systems. This facilitates both constant time combination of taint as well as taint attribution. In this section we will discuss how vectors are handled in our system and how the information they carry is assigned, propagated, and utilized.

We developed V-DIFT in C and designed it to work with 32-bit x86 GNU/Linux executables. V-DIFT forks, attaches, and single-steps through processes using similar APIs as a conventional debugger like GDB. As it single steps, V-DIFT extracts a trace of an attached process in real time, allowing our system to access any memory or address location value in order to calculate address lookups. Each vector is our V-DIFT system contains $n = 200$ floating point values.

Like traditional DIFT systems, V-DIFT assigns taint at designated *sources*. V-DIFT considers any input made through the read system call (`SYS_read`) a source. Additionally, any region of memory can be designated as a source, causing any reads from that memory to be tainted. Memory ranges are designated through a file that V-DIFT reads on startup. To track data provenance, each byte of source data is assigned a random taint vector that is, with high probability, close enough to orthogonal to other taint marks in the system for the results of DIFT analysis to be meaningful.

Conceptually, we want to combine taint vectors such that each vector's length reflects the amount of mutual information the vector carries with any given byte of the taint source. Our logic for combining taint vectors estimates this, but we do not constrain ourselves to any strict information-theoretic model. We desire that if the vectors are perpendicalar, their combined vector length increases, whereas if they are parallel, *i.e.*, the same taint vector, the vector length is unchanged since new information cannot be created by adding information to itself.

V-DIFT *combines* vectors a and b by computing their combination $c = a+b$. Then, the length of c, $\|c\|$, is scaled such that it is equal to
$\min(\gamma, \max(\|a\|, \|b\|) + \min(\|a\|, \|b\|)\,(1 - \mathit{sim}(a,b)^n))$
where n is the number of elements in each vector, γ is a user-defined parameter, and sim is the cosine similarity function.

In addition to γ, V-DIFT has two other user-designed parameters that affect taint propagation: α and β, which affect the propagation of two kinds of indirect flow. α affects taint propagated by the instruction pointer, and β affects taint propagated by address dependencies, *i.e.*, when memory or register locations are used to calculate an address. See Table I for a complete list of how each type of dependency is handled and how the parameters affect each type of dependency.

In V-DIFT, the sink is the trigger for measuring a traced program's taint marks. This measurement can be triggered at any point during program execution. To detect cryptographic keys in memory our sink is any write (`SYS_write`) system call, since we are looking for output bytes heavily influenced by taint marks in each program.

Every time a sink is reached, V-DIFT uses the cosine similarity function to compare the taint vector of each output byte with every initial source vector, creating an m by n matrix, where the rows and columns are the input and output respectively. Each element in the matrix represents the similarity between the particular input and output byte. This allows us to determine how much influence each

Dependency	**Combination**	**Example**
Computation	$\vec{dst} := \vec{src_1} + \vec{src_2} + \ldots$	`add eax ebx` $\rightarrow \vec{eax} := \vec{eax} + \vec{ebx} + \vec{eip} \cdot \alpha$
Copy	$\vec{dst} := \vec{src_1} + \vec{src_2}, + \ldots$	`mov eax ebx` $\rightarrow \vec{eax} := \vec{ebx} + \vec{eip} \cdot \alpha$
Address	$\vec{dst} := (\vec{src_1} + \vec{src_2} + \ldots) \cdot \beta + \vec{drf}$	`lea eax [ebx + 4 * ecx]` $\rightarrow \vec{eax} := (\vec{ebx} + \vec{ecx} + \vec{eip} \cdot \alpha) \cdot \beta + \vec{drf}$
Control	$\vec{flag_1}, \vec{flag_2}, \ldots := \vec{dst}$	`cmp eax esi` $\rightarrow \vec{of}, \vec{sf}, \vec{zf}, \vec{af}, \vec{cf}, \vec{pf} := \vec{eax} + \vec{esi}$
	$\vec{eip} := \vec{eip} + \vec{flag_1} + \vec{flag_2} + \ldots$	`jnz` $\rightarrow \vec{eip} := \vec{eip} + \vec{zf}$

Table I

DEPENDENCY TABLE: $\vec{drf}$ IS THE VECTOR OF THE DEREFERENCED ADDRESS. IN THE COMPUTATION DEPENDENCY CASE $\vec{eip}$ IS ALWAYS ONE OF THE SOURCES ADDED. NOTE THAT EVERY TIME VECTORS ARE COMBINED (REPRESENTED WITH A + IN THIS TABLE) THE VECTOR SCALING DESCRIBED IN SECTION III IS APPLIED.

source byte had on each output byte.

IV. EXPERIMENTAL METHODOLOGY

In this section, we will discuss the experimental setup, from the system running V-DIFT to the details of parameter testing.

A. System Specifications

Tests were run on an Ubuntu 14.04.3 machine running Linux kernel 3.16.0-45. The machine had 256 GB of RAM and an Intel(R) Xeon(R) CPU E5-2637 v2 3.50GHz processor. The large RAM of the machine performing the tests is not a reflection of the specifications needed to run our V-DIFT system but facilitated running multiple tests simultaneously. No test used more than 300 MB of RAM.

B. V-DIFT Parameters

As discussed in Section III there are three user-defined parameters: α, β, and γ. We made these values adjustable to the user because these values may be application-dependent in order to properly propagate taint in the system in a meaningful way. We performed a full factorial experimental design for the parameters to discover the ideal values for our application.

Each cryptography program was executed 30 times, once for each configuration of the following parameter sets for α, β, and γ, respectively: {0.0, 0.0625, 0.125, 0.25, 0.5, 1.0}, {0.0, 0.25, 0.5, 0.75, 1.0}, and {8}.

C. Cryptography Programs

Thorough testing of our system was carried out by testing multiple algorithms across multiple implementations. We chose to test OpenSSL (version 1.0.1), BeeCrypt (version 4.2.1-4), and Crypto++ (version 5.6.1-6), all implementations with available source code. Although there is not a complete overlap of all implementations, the Blowfish algorithm was found in all libraries. We also tested both the ECB and CBC block cipher modes modes where possible. A total of 27 programs were tested (see Table II). Each program reads 128 bytes of data, encrypts the data using one of the algorithms from Table II, and writes the encrypted data to standard out. Each program had the symmetric key in the read-only data section, with typically dozens of kilobytes of other data (such as S-boxes, initialization vectors, locale info, and other program constants) to make locating the key very challenging.

All programs were compiled using the -m32 and -static flags using gcc version 4.8.4. We compile to a 32-bit binary since our implementation only covers 32-bit x86 machine code. The static compilation made running the program much faster and allowed us to easily have less tainted input, as all dynamically loaded libraries would have been tainted otherwise when they were read.

D. Detecting Keys

Key detection is performed by finding regions of memory that had a large effect on the program's output in the trace. Because the entire key affects every encrypted block of data, we expect for it to be highly correlated with the encrypted output. Furthermore, the different bytes of the key will be correlated with each other as they appear in the output. To determine the key region, we use the inner product of the matrix produced from the sink to reveal how correlated inputs are with other inputs at the time of output, as well as the input bytes' effects on the outputs.

Next, we search along the diagonal of the inner product matrix adding up all values in a square window the size of the key. This window size is 16x16 (as the keys were 16 bytes and we taint on the byte level) for all algorithms tested except for DES, as its key size is only 8 bytes. When the number of tainted inputs is much greater than 300, we find the top 300 regions of memory that are most correlated with the output and group together any contiguous memory ranges. These regions are presented to the user (*i.e.*, the reverse engineer using our tool) to examine as likely locations of the key.

E. Timing Tests

The average runtime of each algorithm was calculated by running each program 25 times with $\alpha = 0.0$, $\beta = 0.25$, and $\gamma = 8$, since these parameters produced the best results in terms of locating cryptographic keys. Different modes for an algorithm are considered separate programs. The tests were performed sequentially at random.

V. RESULTS

Figure 1 shows the results for ECB and CBC modes. Specifically, if the cryptographic key is found in the top memory range suggested to the analyst then that is represented by a red square, and so on, as according to the legend.

The most surprising result was that control dependencies were not critical in locating the majority of keys in memory except in the case of OpenSSL DES where control dependencies are necessary. This reveals that implementation is more important than algorithm when detecting keys based on information flow.

Beecrypt	✓	✓							
Crypto++		✓	✓	✓	✓		✓	✓	✓
OpenSSL	✓	✓	✓	✓		✓			
	AES	Blowfish	Camelia	DES	IDEA	RC4	RC5	TEA	Twofish

Table II
CRYPTOGRAPHIC PROGRAMS

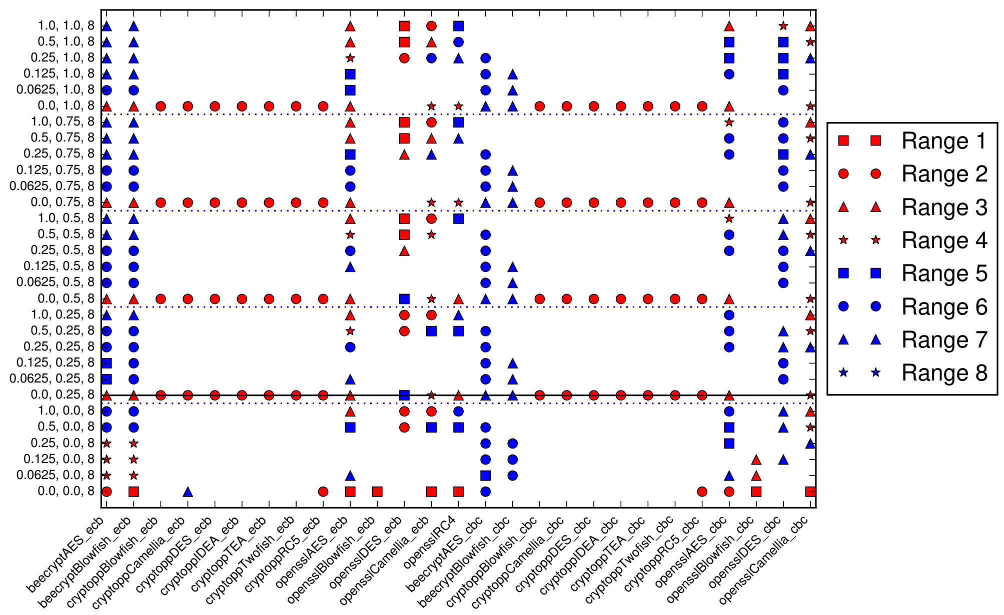

Figure 1. Key detection rankings for parameters α, β, γ.

Our method for detecting keys had varying degrees of accuracy. As seen in Figure 1, the library implementations share more in common with each other than do the algorithms. For example Crypto++ was consistently found in the second memory range for all algorithms. The Beecrypt implementations were close in range as well. OpenSSL did not follow the trend, having caught two and missed two for CBC and only missed one for ECB.

The values chosen as the best parameters (seen as a line in Figure 1) match both ECB and CBC modes, but if the library is known ahead of time the user can adjust parameters accordingly.

Unintuitively, the best parameters for our application for α, β, and γ are 0.0, 0.25, and 8, respectively. The most surprising of these is the α value of 0.0. This means, for our application of DIFT, tracking control dependencies was unnecessary and doing so introduced noise into the system that usually made detecting the key harder. This is surprising because we believed the algorithms that would benefit the most from control dependencies were cryptographic algorithms. The low value of β parameter was also surprising. Our results indicate that address dependencies, even when only using a fraction of their taint, provide the information necessary to locate most keys.

The overall average for all tests was 3.6 seconds. The Blowfish implementations for Crypto++ and OpenSSL were the outliers taking the longest time to complete, taking about 20 and 10 seconds respectively. However, the long runtime does not appear to be due to the algorithm, but the implementation. This is apparent when comparing BeeCrypt's implementation, which completes in about 1 second, to the other two.

Our best results come from $\alpha = 0.0$, $\beta = 0.25$, $\gamma = 8$. Informally, this means "propagate lots of taint for address dependencies, and none for control dependencies." Note that these parameters are specific to the application of locating cryptographic keys. OpenSSL DES (in both ECB and CBC modes) is the only cryptographic algorithm implementation for which control dependencies are necessary to locate the key.

VI. CONCLUSION

We have shown that vectors as taint marks (which we call V-DIFT) is a viable means of storing information in a DIFT system, and gives the information necessary to perform complex tasks such as key location. This is for two reasons: that vectors allow us to distinguish input bytes and that they provide a means for handling indirect flows of information. We have applied V-DIFT to solve a common problem that reverse engineers face: locating cryptographic keys. We found that tracking indirect flows, specifically address dependencies, is important for any DIFT system to produce meaningful results. Also, the performance overhead was very acceptable for offline analysis applications such as reverse engineering. We anticipate, with further research and development, that V-DIFT will lead to whole new ways for reverse engineers to use information flow in their analysis tasks.

Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation under Grant Nos. #1017602, #1314297, #1420716, #1518523, and #1518878. This work was also supported by the DARPA CRASH and Transparent Computing programs. We would like to thank the ARES reviewers and Geoffrey Alexander for valuable feedback.

References

[1] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 621–634, New York, NY, USA, 2009. ACM.

[2] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 169–182, New York, NY, USA, 2012. ACM.

[3] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Understanding data lifetime via whole system simulation. *USENIX*, 2004.

[4] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[5] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can we contain internet worms? In *HotNets III*.

[6] J. Crandall and F. T. Chong. A security assessment of the Minos architecture, 2004.

[7] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[8] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[9] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[10] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 41–60, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] D. D. Hosfelt. Automated detection and classification of cryptographic algorithms in binary programs through machine learning. *CoRR*, abs/1503.01186, 2015.

[12] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[13] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 56–70, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] P. Lestringant, F. Guihéry, and P.-A. Fouque. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 203–214, New York, NY, USA, 2015. ACM.

[15] N. Lutz. Towards revealing attackers' intent by automatically decrypting network traffic. Master's thesis, ETH Zurich, 2008.

[16] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 65–80, Washington, D.C., Aug. 2015. USENIX Association.

[17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[18] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. *MICRO-39*, pages 135–148, December 2006.

[19] A. Slowinska and H. Bos. Pointless tainting? Evaluating the practicality of pointer tainting. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr 2009.

[20] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5):85–96, Oct. 2004.

[21] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, pages 173–184. IEEE Computer Society, 2008.

[22] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 200–215, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] R. Whelan, T. Leek, and D. Kaeli. Architecture-independent dynamic information flow tracking. In *Proceedings of the 22nd International Conference on Compiler Construction*, CC'13, pages 144–163, Berlin, Heidelberg, 2013. Springer-Verlag.

[24] H. Yin, D. Song, M. Egele, and E. Kruegel, Christopher a nd Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM.

[25] R. Zhao, D. Gu, J. Li, and H. Liu. Detecting encryption functions via process emulation and IL-based program analysis. In *Proceedings of the 14th International Conference on Information and Communications Security*, ICICS'12, pages 252–263, Berlin, Heidelberg, 2012. Springer-Verlag.