Pointer Tainting Still Pointless (but we all see the point of tainting)

Asia Slowinska
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
asia@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
herbertb@few.vu.nl

1. INTRODUCTION

A recent article by Michael Dalton, Hari Kannan, and Christos Kozyrakis in this journal ("Tainting is Not Pointless" [6]) criticises the conclusions of our EuroSys'09 paper on pointer tainting ("Pointless Tainting? Evaluating the Practicality of Pointer Tainting" [16]). In a nutshell, our paper shows that pointer tainting has serious problems. Dalton et al. criticise our critique.

In our opinion, the authors are excellent researchers with proven track records in this field. No wonder that we read the article with interest.

Our attitude is simple. Pointer tainting — assuming that it works as advertised — is an extremely powerful technique to detect a variety of attacks. Moreover, we would love to use it. In fact, it was when we did use it that we discovered all sorts of issues which made us abandon the project (and which eventually led to our paper at EuroSys). So if anyone fixes the issues we encountered, we would be the first to welcome it. It would mean that one of the most powerful techniques to detect and analyse malware is not as broken as we thought it was. Great.

So which is it - a crippled technique that can only be used in limited cases, or the powerful detection and analysis technique that most of us thought it was? Or phrased differently: did we change our minds after reading the rebuttal?

The answer is no. We stand by our earlier conclusions. In this article, we explain why. First, we show that the rebuttal paper seriously misrepresents the views expressed in our EuroSys paper. Second, we address specific points and techniques advocated in the rebuttal.

2. POINTLESS TAINTING: WHAT DID WE (NOT) SAY?

Like Dalton, Kannan, and Kozyrakis, we define pointer tainting as a specific form of Dynamic Information Flow Tracking (DIFT). DIFT is a security technique that labels information with tags (such as trusted/untrusted, or public/secret) and controls how such information propogates through a system. As an example, suppose a program copies a chunk of untrusted bytes from array A to array B. In that case, the bytes in B should also be tagged untrusted. However, the exact rules of how to propagate the tags vary from system to system [17]. For instance, some systems propagate tags on computations like `add` and `sub` [12], while others do not [3]. Similarly, we shall see that systems differ in how they handle tainted pointers – and that the difference is crucial.

DIFT may best be described as a 'hot old' security technique. By this we mean that the principles are 'old' – dating back to seminal work by Dorothy and Peter Denning [7] in the 70s, but the area is 'hot' – witness a long string of publications in the last five years in top tier venues (including SOSP, CCS, NDSS, ISCA, MICRO, EUROSYS, ASPLOS, USENIX, Security & Privacy, and OSDI). DIFT is used in programming languages, OS design, protection against high level attacks (such as SQL injections and various web attack) and many others.

Let us start by narrowing down the discussion: we are concerned with the application of DIFT to unmodified binaries. In that case, DIFT tracks the flow of information through a system by means of taint tags. Here is an example. Suppose we tag every byte that we receive from an untrusted source (say the network) as tainted. And whenever that byte is copied or used as an input operand in an operation (e.g., an addition), the result is also tainted. Since we track the untrusted information as well as all values derived from it, we can raise an alert if the data ends up in the wrong place. For instance, in normal programs the program counter should never be tainted.

Of course, rather than tainting untrusted data, we can also taint trusted, privacy-sensitive data and track that such data does not leak to untrusted parties. In either case, because information is tainted, it is common to refer to the technique as 'taint checking', 'taint analysis', or even simply 'tainting'.

2.1 So, did we criticise DIFT?

The rebuttal claims that we do. Even the title, 'Tainting is Not Pointless', suggests that we say it is, or might be. More explicitly, Dalton, Kannan and Kozyrakis write that we state that 'DIFT policies for buffer overflow prevention and malware analysis are rife with false positives and negatives defects.'

It is important to be precise about the claims. Did we really criticise tainting in general? Even without looking at our paper, this seems implausible. Over the years we produced several taint analysis systems [12, 15, 18, 10, 11]. One of these, Argos, is used by many either as a production honeypot, or as a research vehicle [13, 1]. By criticising taint analysis, we would attack much of the work we have done in the past five years. Possible, but not likely.

No, we did not do this. Rather, the paper distinguishes between specific variants of DIFT: basic tainting – which works fine, and pointer tainting – which perhaps does not. Basic tainting is what we described earlier: untrusted data is tagged as tainted and the tag is propagated when the data is copied (either directly, or when it is used as source operand in computations like `add`, `sub`, etc.).

Basic tainting targets attacks that change the control flow of a program. A simple example is shown in Figure (1.a) which represents part of a server program. The server reads a request from a socket in a request buffer. If the attacker provides more than 64 bytes, the `read` will overflow the request buffer and overwrite the `handler` function pointer. When the victim program calls the function in the next line, it really calls the attacker's code instead. Basic tainting detects such attacks by raising an alert (only) on dereferences due to jumps, branches, function calls/returns, when

```
struct req {
 char reqbuf[64];
 void (*handler)(char *);
};

void do_req(int fd,
       struct req *r)
{
 // now the overflow:
 read(fd,r->reqbuf,128);
 r->handler (r->reqbuf);
}
```

(a) control data attack possible

```
void serve (int fd)
{
 char *name = globMyHost;
 char cl_name[64];
 char svr_reply[1024];

 // now the overflow:
 read(fd,cl_name,128);
 sprintf(svr_reply,
        "hello %s, I am %s",
        cl_name, name);
 svr_send(fd,svr_reply,1024);
}
```

(b) non-control data attack possible

Figure 1: Two buffer overflow vulnerabilities. The program in (a) is vulnerable, because if an attacker provides more than 64 bytes to the `read` call, the program will overwrite the `handler` function pointer. When the program subsequently calls the handler, it will really call an address provided by the attacker. The program in (b) is also vulnerable, because if attackers provide more than 64 bytes to the `read` call, they can change the reply string sent by the server - potentially making it leak sensitive information.

the target address is tainted. In this case, it would raise an alert when the handler is called in the last line.

There is nothing wrong with basic tainting. In the paper, we went so far as to write that basic tainting is one of the most reliable methods for detecting control flow diversions and was successfully applied in numerous systems. In summary, we never criticised DIFT or tainting in general. It may be that this was not explicit enough in our paper, but we felt that it might be sufficient to subtitle our paper "Evaluating the practicality of *pointer* tainting', to say that we have no problem with basic tainting, and to state repeatedly, from the abstract to the conclusions, that we talk solely about *pointer* tainting.

2.2 The different uses of pointer tainting (and did we conflate them?)

The main limitation of basic tainting is that it protects against attacks that divert a program's control flow, but not against attacks that do not. Figure (1.b) shows an example. Here the overflow allows the attacker to modify the `name` string and thus the server's reply, potentially causing it to leak information. No code injection, no diversion of the control flow. Clearly, these attacks cannot be detected by basic tainting.

In fact, besides pointer tainting we are not aware of any other technique to detect such attacks reliably on unmodified binaries. Pointer tainting, however, was created expressly for this purpose. Actually, as Dalton, Kannan and Kozyrakis rightly point out, pointer tainting has two very different use cases: stopping memory corruption attacks such as the one above and malware analysis. But since we are on the topic of memory corruption now, let us start with that.

Pointer tainting for detecting memory corruption. Pointer tainting is simply another form of DIFT that builds directly on basic tainting. But it is much stronger. It applies all the taint propagation rules of basic tainting, as well as one additional one. The new rule concerns pointer dereferences, hence the name.

Let us try to make this new rule clear. Basic tainting does not worry about tainted pointers at all other than that it propagates taint as usual (e.g., when the pointer is copied). If a pointer `p` is tainted on a basic tainting system, then a dereference like '`*p = x`' will not raise an alert. Nor will it propagate taint to `*p`.

Pointer tainting, on the other hand, does worry about tainted dereferences. When used to detect memory corruption such as the one in Figure (1.b), it will raise an alert whenever the program dereferences a tainted pointer [2]. In the example, the `name` pointer is tainted by the `read` and dereferenced by the `sprintf`.

Attack detected, problem solved? Not quite. Programs often dereference tainted pointers in a legitimate way. Translating ASCII to UNICODE is a good example. Suppose a server receives an ASCII string from a user and converts it to UNICODE. To do so, it looks up the string's characters in a translation table, constructing an address by adding an index derived from the character to the base of the translation table. The resulting pointer is tainted. Clearly, the dereference is perfectly legal and should not trigger an alert.

Pointer tainting is complex mainly because it must distinguish between legitimate and illegitimate derefences of tainted pointers. A significant part of our paper was devoted to techniques to avoid false positives in the presence of table lookups. More about this issue in Section 3.

Pointer tainting for malware analysis. The other use of pointer tainting is in malware analysis. For instance, to detect whether a program is a trojan keystroke logger, we could taint all characters typed by the user and see whether tainted bytes appear in the address space of the program. To do so, we should not raise an alert when a tainted pointer is dereferenced. After all, a lookup of a character in our conversion table will use a tainted pointer, but it is not malicious. But we should not completely ignore the conversion tables either. Sticking to the previous example, whenever a privacy sensitive character (typed by the user) is translated to UNICODE, the result is sensitive also. In other words, for this application domain, we should *propagate* the taint on pointer deferences.

Unfortunately, pointer tainting in this domain has fundamental problems due to false positives. In our critique, we investigate the problem in detail and show that it is fundamental. The root cause is an undecidable case, where the processor cannot distinguish between two situations where in the one case taint *should* propagate on a pointer dereference, and in the second case it *should not*.

We did not conflate the two use cases. In their rebuttal, Dalton, Kannan and Kozyrakis state – repeatedly – that we conflate DIFT policies for memory corruption with the entirely separate use of DIFT for malware and virus analysis. We disagree. As we summarised above, we distinguish clearly between the two application domains and the different forms of pointer tainting. When pointer tainting is applied to detect memory corruption, we refer to it as LPT or *limited pointer tainting* (as the response to a dereference of a tainted pointer is limited to raising an alert). In contrast, we talk about FPT or *full pointer tainting* when the technique is used for malware analysis, because in that case we go the whole hog: rather than stopping and raising an alert, we propagate taint even more aggressively.

The distinction is made throughout the paper in – literally – every section. We devoted a subsection to each of the two variants to explain in detail what they mean. After that, every analysis, every mitigation technique and every conclusion comes with a label that says whether it pertains to LPT, FPT, or both.

We find it hard to believe that anyone could have missed this. Perhaps the point the authors wanted to make was different, but we can only speculate what that point might be. Again, we stress that we analysed both use cases in the same paper because they have something in common. Something important: they both add to the sound technique of pointer tainting a rule about what to do with dereferences of tainted pointers.

2.3 So, what did we criticise?

We did not criticise DIFT. We did not criticise basic tainting. We did criticise pointer tainting for detecting memory corruption attacks, although we also said that depending on the architecture and operating system pointer tainting one may get it to work. Particularly, we doubted that it could be done for x86 architectures running Windows. Finally, we criticised pointer tainting for malware analysis, FPT, as fundamentally flawed – and yes, rife with false positives and negatives.

More than half of our critique was devoted to FPT. Fortunately, we need not discuss it further in this paper, as Dalton et al. agree that our criticism here is correct. Even so, they mention only the problem of implicit information flow, wheras we tried to show that the problem goes deeper than that. The main problem is false positives, not false negatives[1]. Nevertheless, as we agree that pointer tainting for malware analysis has serious problems, the remainder of this paper will focus on the use of pointer tainting for detecting memory corruption attacks.

3. WERE WE RIGHT TO CRITICISE LPT?

Pointer tainting for detecting memory corruption attacks (LPT) is less cumbersome than using the technique for malware analysis (FPT) and in our critique we even say that the implementation by Dalton et al., is the most promising, most reliable, and most practical of such systems. So why criticise LPT at all? The reason is that issues remain.

Containment techniques. We have seen that the aggressiveness of LPT should be contained in order to prevent false positives on table lookups. There are different ways to do this and we mention four of them in our paper:

1. ebp/esp protection,
2. detecting and sanitising table accesses,
3. bounds check recognition (BCR),
4. pointer injection detection (PI).

Strangely, Dalton, Kannan, and Kozyrakis talk about the first three only (and agree that they do not work) and then present the fourth one as their rebuttal. We want to emphasise that we also discussed pointer injection in our paper! We devote almost a full column of text to PI. We even said that things are mostly fine on SPARC/Linux [5]. We also said that it will be a challenge to port it to architectures like the x86 and operating systems like Windows.

Portability. To see where portability issues arise, let us briefly look at pointer injection. The main idea is that we identify the legitimate pointers in the system. If the program dereferences a legitimate pointer, everything is fine. However, a dereference of a pointer that is not a legitimate pointer is not okay. Typically, it means that an attacker is trying to inject a pointer. Thus, at startup time, PI marks all pointers to statically allocated memory as pointers. At runtime, it marks all return values of system calls that dynamically allocate memory as pointers. Finally, PI tracks the propagation of pointers and also the propagation of untrusted (tainted) data, and raises an alert whenever a dereference is made of a tainted value that was not explicitly marked as pointer.

[1]However expensive, it may be possible to prevent these false negatives by raising alerts as soon as taint appears in the process' address space – before the malware can launder it.

In principle, we like the idea. PI elegantly solves the problem of table lookups. It requires two things. First, we should not miss any pointers during the marking process, as this would lead to false alarms. Second, we should try our best to mark only real pointers as pointers, because erroneously marked values may lead to false negatives.

A port of PI to Windows is hard, for instance because finding pointers at startup time is complicated. In the case of Linux, the authors use hardcoded constants in header files for a variety of memory regions [5]. No such header files are available for Windows. Of course, one may scan a binary and conservatively mark values as pointers if they *could* be pointers, but getting it wrong is not without consequences. In particular, they may lead to false negatives.

As one cannot identify kernel heap regions, the authors suggest to treat any value that points into kernel address space conservatively as a pointer. This is also what we suggested in our paper. Again, doing so increases the number of false negatives. Moreover, we do not see how one could still protect kernel memory in such a situation. The ability to protect kernel and userspace was one of the selling points of Raksha [5].

A port to x86 is also hard, because scanning instructions to look for pointer manipulations is harder on x86 than on SPARC. The reasons are that static disassembly is undecidable and that pointer manipulations are hard to distinguish from scalar handling. As we mentioned above, overestimation leads to false negatives.

In general, though, PI is not easy to get right on any architecture. This is not to say that it is impossible, but subtle issues arise. Let us consider the rules for PI proposed by the authors of the rebuttal in [5]. The authors painstakingly map out the rules for DIFT propagation so as to avoid false positives. For instance, when two values are added or subtracted and either of them is marked as pointer, the result will also be a pointer. This looks like the right thing to do, even though one might mistakenly classify a distance between two pointers (a subtraction of two pointers) as a pointer itself. We do not consider this very serious. In general, false negatives are less important than false positives.

The point is that it is easy to make mistakes. Some *are* serious. For instance, the authors claim that other ALU operations like multiply or shift should not be performed on pointers. So whenever they encounter such operations, the result is marked as 'not a pointer'. This makes sense in most cases, but not always. For instance, here is a snippet of assembly of the `wget` download program:

```
8069253: lea    0x24(%esp),%eax  # %eax is a pointer
8069257: add    $0xf,%eax
806925a: shr    $0x4,%eax        # shift right
806925d: shl    $0x4,%eax        # shift left
```

In this example, we see that some applications do perform shifts on pointers – for alignment for instance. PI with the above rules would mistakenly conclude that `%eax` is not a pointer, leading to false positives. It is fairly easy to remedy the rules to make them cope with this particular problem, but it would again increase the number of false negatives. And it would be difficult to determine when the rules are really foolproof.

To quantify the problem of possible false negatives, we implemented a pointer tainting DIFT system exactly like Raksha [5], but rather than using the Linux header files, we identify pointers by value - as proposed by Dalton et al. The taint propagation rules are the same as Raksha's (with the sole exception that we fixed the issue with the shift operation described above). We then measured how many stack values were erroneously tagged as pointers. For a binary like the `wget` download program, 14305 out of 215758

or 6.6% of all values tagged as pointers are not really pointers at all - they just happen to point to valid memory areas. Moreover, 13.1% of all functions executed have at least one such a misclassified value. In our opinion, such numbers are not negligible and some of the 'pointers' may be used by attackers.

In summary, we like PI and still believe it is one of the most promising techniques for containing taint propagation in pointer tainting. However, as demonstrated above, it is not easy to get it right. It is even harder to do so on an architecture like x86 and for closed source operating systems like Windows. We do not say that it is impossible. Also, if anyone can do it, it would be the group of Dalton, Kannan and Kozarykis. And it would be great if they did it. All we say is that it will not be easy.

Implicit information flow. According to Dorothy and Peter Denning [7], an implicit flow of information from x to y occurs whenever a statement specifies a flow from some arbitrary z to y, but execution depends on the value of x. In general, all conditional structures generate implicit flows. In our context, an implicit flow means that even though there is no direct assignment of a tainted value to a variable, the value of a variable is completely determined by the tainted value [14]. Conditional statements such as "`if (x==0) y=1; else y=2;`" are the best-known examples of implicit information flow. Suppose `x` is tainted. As there is no assignment of `x` to `y`, the latter variable will not be tainted, even though it is completely determined by the value of `x`.

Dalton, Kannan and Kozyrakis acknowledge that implicit information flow is a problem for DIFT, but only for malware analysis. We already argued that the problems for malware analysis go much deeper. Besides false negatives, we demonstrated an abundance of false positives. In this section, however, we are concerned with their rebuttal that implicit information flow is only a problem in malware analysis: "No known memory corruption attacks in the real world rely on implicit information flow". They also say that none of these criticisms apply to DIFT policies for memory corruption attacks. In their opinion, "policies for memory corruption prevention need only track common flows of information such as data movement".

We disagree and present an example – from their own paper [4] – of a realistic memory corruption attack that relies on implicit information flow to escape tainting. It does not involve a condition in the form of an `if` statement, but there is no reason it should. Instead, it uses a jump table that is indexed with a tainted value. Moreover, it is a common attack in the sense that existing format string errors are susceptible to it.

Again, we only aim to show that these issues *do* exist. They lead to false negatives. But as we mentioned earlier, we consider false negatives less important than false positives. And as in our EuroSys paper, we believe that pointer tainting for memory corruption may well be useful, as long as one accepts false negatives (and it remains to be seen if/how it works on x86/Windows combinations).

We will start with a description of the attack. The `printf` family of functions, if used carelessly, allows attackers to write an arbitrary value to an attacker-specified address – this is known as a format string attack. The problem stems from the use of the `%n` format specifier, which causes `printf` to write the number of bytes printed to an address specfied as an argument (on the stack).

In a nutshell, a format string attack works as follows. A sloppy programmer writes code that reads a string `str` of user input and prints it for instance as follows: `printf(str)`. If the input string is a normal string, all is well. However, instead of a normal string, an attacker may provide as input a format specification, such as `"hello%n"`. What happens is that `printf()` will interpret the format and display characters accordingly. In this case, it will print `hello` and then store the value 5 (the number of characters displayed so far) in the memory location specified by the next argument of the `printf()`. In this example, we did not specify a second argument, but `printf()` does not know that. It will simply interpret the 4 bytes above the format string on the stack as an address and write the value 5 to that address.

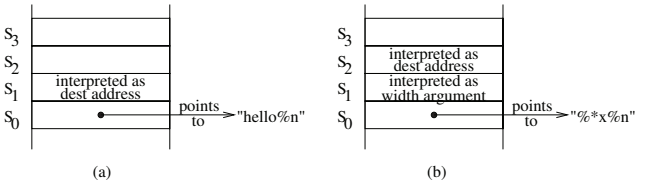

Figure 2: Stack layout during our format string attacks

The stack is sketched in Figure (2a). The address to the format string provided by the attacker is the first argument to `printf()` (at address s_0). There is no real second argument, so `printf()` will interpret the value at s_1 as the second argument (the address where it should store the number of printed characters).

Since the format string provided by attackers cannot have an arbitrary length, attackers often use a constant field width specifier which determines the minimum number of characters to be printed. For instance, `printf("%50x%n",x,&y)` will print the value of `x`, padded with spaces to 50 characters, and then assign the number of printed bytes (50) to `y`. Using the field width specifier, attackers can write an arbitrary value to an arbitrary location, while keeping the format string short. Unfortunately for the attacker, in most DIFT systems the value will be tainted: it is calculated by the `printf()` function by adding up the number of characters printed and this calculation uses the field width specifier.

This is bad for attackers. However, there is another way. And this time, the value is not tainted. Rather than providing the field width directly, the `printf()` family permits us to specify the field width as `*`, which means `printf()` will interpret the next argument on the stack as the (integer) field width. In other words, the field width specifier is now read from the stack. To illustrate this, Figure (2.b) shows the stack when the attacker provides the format string`"%*x%n"`. Again, `printf()` will find a pointer to the format string as its first parameter and look for the remaining arguments on the stack. Thus, the value at s_1 will be interpreted as a field width specifier, and the value at s_2 as the destination address.

It is not very difficult to pick a specific value on the stack to use. For instance, by providing the format string `"%x%*x%n"`, an attacker forces the function to use the value at s_2 as the field width specifier and the value at s_3 as the address. When carefully chosen, the field width specifier will not be tainted. It turns out that in many implementations one can specify directly which value on the stack one wants to use, by using positional parameters. However, this is beyond the scope of this paper. We refer interested readers to the excellent paper by Dalton et al. [4]. The important message is that these attacks are real. A classic example of an exploit that works this way is LSD's attack on the IRIX telnet daemon [8].

Also, we leave it as an exercise for the reader to construct more sophisticated attacks that 'set' the address of the memory location to modify, and then 'set' the value at that address. If attackers are careful enough, they can write an arbitrary non-tainted value at a location that is pointed to by a non-tainted value on the stack. Even advanced detection techniques like PI would not detect such attacks. The authors of Raksha therefore propose to handle such functions separately. We agree that this is probably the only thing

we can do.

Again, in this section we are not concerned with detecting such attacks. We merely want to show that they are real and that they are examples of implicit information flow. We will now zoom in on the implicit flow aspect.

Intuitively, we already see that this is an implicit flow of information as an untainted value is completely dependent on a tainted value. Let us look at the examples in a little more detail to see what form of implicit information flow it is[2]. To process a format string, the `vfprintf()` function iterates over the format string's characters, constantly jumping to appropriate code blocks by way of the `step0_jumps` table. For instance, when it encounters the '`*`' field width specifier, it will jump to the corresponding code. The jump table is thus indexed using tainted characters. In essence, this is a very efficient implementation of the following (pseudo-)code:

```
if (character == '+')
    do_plus();
else if (character == '%')
    do_percent();
 .
 .
 .
else if (character  == '*')
    width = read_from_stack();
    do_width_asterics ();
 .
 .
```

In this code, `character` is tainted, but since the taint is not propagated through the control flow, the `width` value read from the stack is untainted. Thus its usage is not tracked, and possible attacks go undetected.

In general, implicit information flows are hard to track. They require analysis of both the taken and the non-taken paths in branches. Even in higher-level managed languages like Java researchers have failed to get it correct for all cases [9].

4. CONCLUSIONS

In this paper, we argued several things. First, that the rebuttal by Dalton, Kannan, and Kozyrakis misrepresents our views and the views expressed in our Eurosys'09 paper. This is quite unfortunate, because casual readers may think that we carelessly dismissed an important security technique. We did nothing of the sort. Second, that some of the solutions in the rebuttal are expressly discussed in our paper. Proposing them as a rebuttal suggests that we were not aware of them, which is not the case. Third, that we agree that pointer tainting for malware analysis is problematic, but that our criticism goes beyond false negatives due to evasion by the malware, and expressly includes false positives also. Fourth, that implicit information flow is problematic for detecting memory corruption also.

Thus, the disagreement that remains concerns the technique of pointer injection. In our Eurosys'09 paper we argued that this is a promising and reliable method (and one that may indeed be useful), but that it would be difficult to port to the x86 architecture and the Windows operating systems. Dalton, Kannan, and Kozyrakis disagree and suggest solutions for some of the problems. Note that we are talking about a minor part of both our original paper and the rebuttal. In this paper, we argue again that it really is not easy to get PI right, especially on x86/Windows combinations, although we do not say that it is impossible! In light of the above, we stand by our analysis.

[2]The analysis is based on `libc-2.10.1`.

5. REFERENCES

[1] J. Berg, E. Teran, and S. Stover. investigating argos. *USENIX ;LOGIN:*, pages 29–35, 2008.

[2] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *DSN '05*, Washington, DC, USA, 2005. IEEE Computer Society.

[3] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05*, 2005.

[4] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing hardware architectures for security. In *WDDD'06*, June 2006.

[5] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th Usenix Security Symposium*, July 2008.

[6] M. Dalton, H. Kannan, and C. Kozyrakis. Tainting is not pointless. *SIGOPS Oper. Syst. Rev.*, 44(2):88–92, 2010.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[8] LSD. Irix telnet daemon exploit `irx_telnetd.c` and explanations. `http://www.securityfocus.com/templates/archive.pike?list=1&mid=75864`.

[9] S. Nair. *"Remote Policy Enforcement Using Java Virtual Machine"*. PhD thesis, Vrije Universiteit Amsterdam, January 2010.

[10] G. Portokalidis and H. Bos. Eudaemon: involuntary and on-demand emulation against zero-day exploits. In *ACM SIGOPS EUROSYS*, 2008.

[11] G. Portokalidis, P. Homburg, N. Fitzroy-Dale, K. Anagnostakis, and H. Bos. Protecting smart phones by means of execution replication. Technical Report IR-CS-54, Vrije Universiteit Amsterdam, September 2009.

[12] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, 2006.

[13] N. Provos and T. Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley Professional, 2007.

[14] H. J. Saal and I. Gat. A hardware architecture for controlling information flow. In *ISCA '78*, pages 73–77, New York, NY, USA, 1978. ACM.

[15] A. Slowinska and H. Bos. The age of data: Pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *ACSAC*. IEEE Computer Society, December 2007.

[16] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[17] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*. ACM, 2004.

[18] M. Valkering, A. Slowinska, and H. Bos. Tales from the crypt: fingerprinting attacks on encrypted channels by way of retainting. In *Proc. of 3rd European Conference on Computer Network Defense (EC2ND)*, October 2007.