Security bugs in embedded interpreters

Haogang Chen   Cody Cutler   Taesoo Kim   Yandong Mao
Xi Wang   Nickolai Zeldovich   M. Frans Kaashoek

MIT CSAIL

Abstract

Because embedded interpreters offer flexibility and performance, they are becoming more prevalent, and can be found at nearly every level of the software stack. As one example, the Linux kernel defines languages to describe packet filtering rules and uses embedded interpreters to filter packets at run time. As another example, the RAR archive format allows embedding bytecode in compressed files to describe reversible transformations for decompression. This paper presents an analysis of common pitfalls in embedded interpreter implementations, which can lead to security vulnerabilities, and their impact. We hope that these results are useful both in augmenting existing embedded interpreters and in aiding developers in building new, more secure embedded interpreters.

1 Introduction

Many systems offer customization by allowing third parties to download executable extensions [9]. For flexibility and portability reasons, these systems often define an instruction set, in the form of bytecode, and implement an *embedded interpreter* to run downloaded bytecode. For performance reasons, they may translate bytecode into machine code before execution [8], using a just-in-time (JIT) compiler. One example is the Berkeley Packet Filter (BPF) [15]. An OS kernel accepts packet filters from user space, in the form of bytecode. The kernel implements an interpreter to execute the BPF bytecode against network packets and drop unwanted ones.

Embedded interpreters raise interesting security concerns. First, many real-world systems do *not* adopt sandboxing techniques such as process isolation [20] or software fault isolation [28] for embedded interpreters, possibly due to performance considerations. Consequently, a compromise of the interpreter is likely to lead to a compromise of the host system as well. Second, embedded interpreters often validate untrusted bytecode using ad-hoc rules, which is error-prone. Third, the bytecode (e.g., BPF filters) usually accepts input data (e.g., network packets), in which case both the bytecode and its input data may be *untrusted*, thereby exposing the host system to a wider range of attack vectors. An embedded interpreter must defend against both malicious bytecode and malicious input data; failing to do so can lead to a compromise.

This paper investigates the security implication of deploying embedded interpreters in systems. The first contribution is a case study of how systems use embedded interpreters in practice. As we will show in §2, they are surprisingly widespread. For example, the Linux kernel alone hosts multiple embedded interpreters, which are used for packet filtering [15], system call filtering [7], network monitoring [13], and power management [11]. The Clam AntiVirus (ClamAV) engine runs an interpreter to identify viruses with complex signatures [30], and also runs another interpreter for inspecting RAR files, which can contain bytecode for decompression [17]. Even a TrueType font file can contain bytecode for rendering; the infamous JailbreakMe exploit took advantage of a vulnerability in the TrueType interpreter via a crafted font file, leading to privilege escalation on iOS devices [24].

The second contribution of this paper is a case study of attack vectors for embedded interpreters. In the worst-case scenario, an adversary controls both the bytecode and the input data. To ensure safety, an embedded interpreter may have to regulate memory access, handle undefined operations (e.g., division by zero), and avoid resource exhaustion (e.g., infinite loops). As an example, failing to defend against malicious code and data in BPF can lead to kernel vulnerabilities, the consequences of which include crashes, information leaks, and even arbitrary code execution. §3 illustrates such vulnerabilities and their attack vectors.

Building on the case study of embedded interpreters, §4 summarizes state-of-the-art defense techniques and provides security guidelines. §5 discusses research directions for improving the security of embedded interpreters. For example, testing embedded interpreters is challenging because one must consider corner cases in both the bytecode and the input to it. Finally, §6 concludes.

2 Embedded interpreters

Figure 1 summarizes embedded interpreters discussed in this paper. Each interpreter is designed to execute bytecode of a hypothetical machine that aims at solving problems in a specific domain, as detailed next.

Linux Socket Monitoring Interface (INET-DIAG). INET-DIAG is designed for socket monitoring. For example, consider the following ss command [13], which monitors sockets with source port in range $[21, 1024)$:

```
ss 'sport >= :21 and sport < :1024'
```

The ss command generates the following bytecode and submits it to the kernel:

```
     sge    21, L1, rej # if (sport >= 21)   goto L1
                        # else               goto rej
L1:  sge  1024, L2, acc # if (sport >= 1024) goto L2
                        # else               goto acc
L2:  jmp   rej
acc: nop                # accept
rej:                    # reject
```

The INET-DIAG interpreter executes the bytecode in the kernel and returns a list of sockets for which the execution reaches "accept" (nop).

The INET-DIAG bytecode supports comparisons and forward jumps, but not backward jumps, to avoid loops. Therefore, the interpreter must check that every jump offset must be positive. Failing to reject bytecode that violates this invariant will lead to infinite loops (§3.1).

Berkeley Packet Filter (BPF). BPF is used in many Unix-like OS kernels to filter link-layer packets [15]. Recently, the Linux kernel added support for system call filtering via BPF [7]. BPF uses a register-based virtual machine, which consists of two registers and a small scratch memory. The BPF interpreter executes bytecode to check input data, and returns a boolean result. The Linux kernel also includes a JIT compiler for BPF [4]. BPF supports integer arithmetic, logical operations, branches, and forward jumps. One example of a malicious operation that BPF should prevent is division by zero; failing to prevent it leads to a crash (§3.2). Since the BPF interpreter supports a memory region, failing to zero the memory before use will leak information of the host system (§3.3).

ACPI Machine Language (AML). AML [11] defines ACPI control methods, which instruct the OS kernel how to respond to certain power management events. Because AML bytecode is usually loaded from the firmware, it is considered trusted and allowed to do anything, including accessing arbitrary memory, performing device I/O, and invoking any functions in the kernel. The Linux kernel, however, allows user-space applications to override some control methods. If the permission checks in this override mechanism are insufficient, an adversary can inject and execute arbitrary code in the kernel (§3.4).

Bitcoin. Bitcoin uses an interpreter to define transactions over its network [1]. The Bitcoin interpreter is a stack machine. The input, such as public keys, are embedded in its bytecode as constants. The interpreter returns whether a transaction is valid or not. Besides conditional branches, arithmetic, and logical operations, the interpreter also supports many string and cryptographic operations to validate a transaction.

ClamAV. The ClamAV antivirus engine uses LLVM-based bytecode as signatures for polymorphic malware [30]. LLVM is a register machine that supports arithmetic and logical operations, branches, jumps, and function calls. ClamAV's interpreter and JIT engine sandboxes the bytecode so that all pointers are bounds-checked, loops have timeouts, and external function invocations are disallowed.

TrueType and Type 2 Charstring. TrueType defines a hinting language for rendering fonts [27]. The bytecode manipulates control points in the font outline. The TrueType VM is a stack machine that supports arithmetic and logical operations, branches, loops, and function calls. It also has opcodes to move, align, and interpolate points in various ways. The Type 2 Charstring VM defined in Adobe Type 2 Font [12] is similar to TrueType, except that it does not allow jumps and loops.

Universal Decompressor Virtual Machine (UDVM). UDVM [19] is used by Wireshark, a packet analyzer, to decompress certain stream protocols. UDVM provides 64 KB of memory and a call stack. It supports arithmetic, logical, and some string operations. These instructions can reference memory directly. UDVM also supports branches and loops. UDVM imposes cycle limits on bytecode based on the length of the input data.

interpreter	arith.	br.	loop	func.	ext.	JIT	bytecode source	input source	description
INET-DIAG		Y					user space	kernel	network monitoring [13]
BPF	Y	Y				Y	user space	network	packet & syscall filtering [7, 15]
AML	Y	Y	Y	Y	Y		firmware	device	power management [11]
Bitcoin	Y	Y					network	network	digital currency [1, 16]
ClamAV	Y	Y	Y	Y		Y	file & network		antivirus engine [30]
TrueType	Y	Y	Y	Y			file & browser		font rendering [26, 27]
Type 2	Y	Y		Y			file & browser		font rendering [12]
UDVM	Y	Y	Y	Y			file	network	universal decompressor [19]
RarVM	Y	Y	Y	Y	Y		file	file	decompressor filter [17]
Pickle				Y	Y		file & network		data serialization [18]

Figure 1: Summary of embedded interpreters and their features. Here "arith." means arithmetic operations; "br." means conditional branches and forward-only jumps; "func." means user-defined functions; "ext." means external function calls; "JIT" indicates whether there is a known JIT implementation.

RarVM. RAR files can contain RarVM bytecode that performs some reversible transformation on input data to increase redundancy [17]. RarVM is an x86-like register machine. It provides 8 registers and 64 KB of memory with a stack at the top. It supports arithmetic and logical operations, branches, loops, and function calls. RarVM sandboxes its bytecode in a way similar to ClamAV, but allows some external function calls.

Pickle. The Python standard library provides the Pickle [18] module for serializing and deserializing Python objects. The Pickle protocol defines a stack interpreter, and executes bytecode to deserialize Python objects. Pickle's input data is embedded in its bytecode program. The Pickle interpreter provides opcodes to manipulate Python objects such as dictionary and list, as well as opcodes to load and invoke external Python libraries. Therefore, inappropriate use of the Pickle library often leads to privilege escalation attacks, or arbitrary code execution in remote machines, as described in Figure 2.

3 Vulnerabilities in embedded interpreters

An embedded interpreter that receives bytecode from an untrusted source must validate it before execution. In addition, if input data to the bytecode is also untrusted, the interpreter must execute the bytecode defensively. This section illustrates common vulnerabilities found in embedded interpreters that fail to do so. Figure 2 summaries the vulnerabilities studied in this section.

3.1 Resource exhaustion

When an embedded interpreter runs bytecode, it should be able to control resource consumption. If the bytecode supports jumps [11, 17, 30], it is possible to construct backward jumps that lead to infinite loops; the interpreter should constrain such jumps. Similarly, if the bytecode supports subroutines, the interpreter should guard against infinite recursion that will lead to stack overflows.

Figure 3 shows a vulnerable code snippet, from the INET-DIAG interpreter in the Linux kernel, as well as the corresponding patch. The interpreter increases the instruction pointer by `inet_diag_bc_op->yes` when the current instruction evaluates to true. To forbid backward jumps, the code defines "`yes`" as an unsigned integer; but it forgets to validate that `yes` is non-zero, leaving the interpreter vulnerable to infinite loops.

A general way to constrain infinite loops or infinite recursion is to limit the number of executed instructions and the depth of nested function calls. Every time an instruction is executed or the depth of call stack is increased, the counter is incremented. The interpreter aborts the execution when the counter reaches a limit. In complex interpreters, however, resource consumption can happen in many places in the code. Implementing ad-hoc watchdog counters becomes non-trivial and error-prone, as we can find in CVE-2010-2286 and CVE-2011-3627. Without careful plans for restricting resources, it is hard to avoid DoS in embedded interpreters.

3.2 Arithmetic errors

Embedded interpreters that provide arithmetic instructions are prone to arithmetic errors. These errors may come from unexpected arithmetic behaviors of the target machine. For example, when handling a signed division instruction, an interpreter should check that the divisor is non-zero, and that the quotient does not overflow (i.e., not $\texttt{INT\_MIN}/-1$); otherwise, the division might trigger a machine exception and potentially result in program termination.

Checking for overflow conditions can be tricky and error-prone [29]. Figure 4 shows the ClamAV interpreter

Vulnerability type	Vulnerabilities	Virtual machine	Software
Resource exhaustion	CVE-2010-2286	UDVM	Wireshark
	CVE-2010-3880	INET-DIAG	Linux
	CVE-2011-2213	INET-DIAG	Linux
	CVE-2011-3627	LLVM	ClamAV
Arithmetic errors	CVE-2010-5137	Bitcoin	bitcoind
	CVE-2007-3725	RarVM	ClamAV
Information leak	CVE-2010-4158	BPF	Linux
	CVE-2012-3729	BPF	Apple iOS
Arbitrary code execution	CVE-2010-4347	AML	Linux
	CVE-2011-1021	AML	Linux
	CVE-2011-2520	Pickle	Fedora firewall config
	CVE-2012-4406	Pickle	Openstack-Swift
Memory corruption	CVE-2010-2995	UDVM	Wireshark
	CVE-2010-2520	TrueType	FreeType
	VU#662243	RarVM	Sophos Antivirus

Figure 2: Examples of vulnerabilities of embedded interpreters.

```
struct inet_diag_bc_op {
    unsigned char   code; // opcode
    unsigned char   yes;  // instruction length
    unsigned short  no;   // conditional jump offset
};
const void *bc = bytecode;
while (len > 0) {
    struct inet_diag_bc_op *op = bc;
    ...
+   if (op->yes < min_len // min_len is at least 4
+       || op->yes > len + 4 || op->yes & 3)
+       return -EINVAL;
    bc += op->yes;
    len -= op->yes;
}
```

Figure 3: The patch that fixes CVE-2011-2213. The vulnerability was missing validation of INET-DIAG instructions in the Linux kernel, which results in infinite loop if `op->yes` is zero.

```
case OP_SDIV:
{
    int64_t a = BINOPS(0); // dividend
    int64_t b = BINOPS(1); // divisor
    if (b == 0 || (a == -1 && b == INT64_MIN))
        return CL_EBYTECODE;
    value->v = a / b;
    break;
}
```

Figure 4: Incorrect handling of the signed division opcode in ClamAV's interpreter. The correct check to avoid signed division overflow should swap `a` and `b` (i.e., `a == INT64_MIN && b == -1`).

attempting to verify division operands but mistakenly mixing up the dividend and divisor, allowing a denial-of-service attack. Similarly, a proposed patch to "fix" the DragonFlyBSD bug #1748 [6] was incorrect, allowing specially crafted BPF filters to cause a kernel panic.

Bit shifts can also lead to arithmetic errors. For example, CVE-2010-5137 shows that a missing check of an oversized shift amount causes Bitcoin to crash on some machines when processing a transaction containing a left-shift opcode.

3.3 Information leak

When an embedded interpreter exposes unintended information to the bytecode in its input or in its execution environment, malicious users can extract this information by crafting bytecode and observing the result of its execution. This type of vulnerability often happens when an embedded interpreter simulates register files, scratch memory, or uses a stack for executing the bytecode. If these buffers were not properly initialized, malicious bytecode could output uninitialized values and obtain sensitive information about the host system.

Figure 5 shows code from the BPF interpreter in the OpenBSD kernel, which did not zero out the `mem` array on the kernel stack. A malicious filter might return the value of an unused memory slot as the number of accepted bytes, potentially exposing kernel randomness to userspace, and thereby breaking security mechanisms that depend on randomness, such as Address Space Layout Randomization (ASLR) [23]. A similar vulnerability (CVE-2012-3729) was discovered in the Linux kernel.

3.4 Arbitrary code execution

Enabling external calls breaks the isolation between the interpreter and the host system, and can lead to arbitrary code execution in the host system if not carefully designed. Unless both the bytecode and the input come from trusted

```
 u_int bpf_filter(pc, p, wirelen, buflen)
 {
     ...
     u_int32_t mem[BPF_MEMWORDS];  // scratch memory
+    bzero(mem, sizeof(mem));
     ...                           // run the filter
 }
```

Figure 5: CVE-2012-3729: OpenBSD BPF filters loading from uninitialized scratch memory (mem array) could leak sensitive information from the kernel stack. The patch initializes it with bzero.

```
 while ( ip < limit ) {
     CFF_Operator  op = *ip++;
     FT_Fixed   *args = decoder->top;
     ... // handling op, which may change args
     decoder->top = args;
+    if (decoder->top - stack >= CFF_MAX_OPERANDS)
+        goto Stack_Overflow;
 }
```

Figure 6: CVE-2010-1797: Stack-based buffer overflow in FreeType. The Type 2 Charstring interpreter forgets to validate the stack pointer after processing an operator.

sources, the embedded interpreter should have a clear plan for isolating the execution of the bytecode.

For example, Linux's AML interpreter can invoke any kernel functions. Usually this is not a problem because the AML bytecode comes from firmware in trusted hardware. However, when a kernel developer accidentally exposed the custom control method interface to unauthorized users, the lack of isolation in the AML interpreter resulted in a severe vulnerability (CVE-2010-4347).

Another example is the notorious Pickle module in Python. The Pickle interpreter defines GLOBAL and REDUCE opcodes that provide the capability for importing and running arbitrary builtin Python libraries. Many Python developers are unaware of the security implications, and use Pickle to deserialize objects from untrusted sources, thereby permitting remote code execution [25], such as CVE-2012-4406 and CVE-2011-2520.

3.5 Memory corruption

Many embedded interpreters are written in an unsafe language, and programming errors can lead to memory corruption errors. Examples include mishandling of the execution stack (Figure 6), omitting bounds checks when executing string operations, etc. We do not dive further into these problems since they are the result of implementation errors, as opposed to design issues specific to embedded interpreters.

3.6 JIT spraying

In order to speed up execution, some embedded interpreters [15, 30] use a JIT engine to compile their bytecode into native instructions. JITted bytecode introduces a new attack vector called JIT spraying [2, 21], in which attackers encode shell code as constants in benign-looking bytecode. These constants can be strung together to form code gadgets that facilitate return-oriented programming [22].

For example, when specially constructed BPF bytecode is compiled and loaded into the kernel, attackers can trigger another memory corruption vulnerability, and jump into the middle of the JIT-compiled code, where the attacker-controlled constants are interpreted as native machine instructions.

JIT spraying works for two reasons. First, JITted bytecode usually resides in pages which are concurrently writable and executable. This can effectively disable existing protection techniques, such as DEP (Data Execution Prevention) and SMEP (Supervisor Mode Execution Protection) [14]. Second, JIT engines transform bytecode from untrusted sources in a rather predictable way. Attackers can force the JIT engine to generate many copies of the same gadget throughout the memory, diminishing the protection provided by ASLR [23].

4 Security guidelines

§3 shows that it is non-trivial to design and implement a secure embedded interpreter. In this section, we suggest some guidelines for improving the security of embedded interpreters.

Process isolation. When performance is *not* a critical factor, one can consider securing an embedded interpreter via process isolation. For example, web browsers such as Chrome [20] isolate the execution of JavaScript for different web pages in separate OS processes. As another example, RarVM bytecode can execute in a dedicated process because the overhead of creating a process is amortized over the cost of decompressing a RAR file.

This approach has several limitations. First, it incurs performance overhead due to inter-process communication (IPC), which makes it less appropriate for performance-critical applications such as packet filtering. Second, it cannot prevent semantic bugs, where a compromised interpreter can produce wrong results (e.g., incorrect files decompressed from RarVM). Third, process isolation is difficult to apply inside OS kernels.

Limiting resource consumption. If an embedded interpreter runs within the context of the host system, it is critical to ensure that the execution of bytecode does not exhaust the host system's resources. In particular, in order to provide jumps or subroutines, interpreters should have a plan for monitoring and constraining the processor time and memory that bytecode utilizes. Whenever the bytecode violates the resource utilization policy, the interpreter should be able to detect and reclaim allocated resources, thereby avoiding DoS attacks on the host system. As illustrated by INET-DIAG and BPF, allowing only forward jumps, or limiting the number of executed instructions or execution time is necessary to constrain the resource usage of embedded interpreters.

Limiting feature sets. One interesting observation is that the more expressive a bytecode design is, the more invariants an interpreter implementation must maintain, which consequently enables a wider range of possible attack vectors. A developer designing an embedded interpreter should consider the trade-off between flexibility and security in their design. For instance, Bitcoin chose to disable certain instructions [1] to reduce attack vectors after several arithmetic errors were discovered in its implementation (see §3.2). In this vein, INET-DIAG, which does not support arithmetic operations by design, is more immune to arithmetic errors such as division by zero. If the embedded interpreter needs to provide a Turing-complete instruction set, extra care should be taken in its implementation.

Limiting calls to host. When a bytecode program needs to interact with or to process external input given by the host machine, interpreters should define a clean interface between them. For example, in designing a new Pickle protocol, one could define a set of safe Python libraries that Pickle programs can invoke. Unlike Pickle, BPF bytecode has a clean input and output interface to a host machine; it takes a packet as the input, and outputs a bit indicating whether the input packet is filtered or not.

5 Research problems

Testing embedded interpreters. Symbolic testing tools such as KLEE [3] and SAGE [10] explore code paths and construct input data to trigger bugs. To make those tools effective in testing bytecode interpreters with high coverage, one needs to consider how to construct not only untrusted input data, but also untrusted bytecode, as control flow decisions in interpreters are highly dependent on the bytecode, and symbolically exploring all the paths can easily lead to path explosion.

For example, a naïve approach is to generate a random bytecode program for testing. However, this bytecode program is likely to be malformed and simply rejected by the interpreter, and thus is ineffective for exploring code paths of the interpreter's implementation. An alternative approach is to develop a specific generator tailored for each interpreter. How to effectively derive bytecode instances that have both high coverage and little path duplication remains an open problem.

Moreover, the prevalence of JIT compilation poses another intriguing research problem: how to systematically test the correctness of JITted bytecode? The traditional symbolic testing paradigm does not fit JIT schemes well because the generated native code, which itself is the output of the symbolic bytecode, also needs to be symbolically executed. Scheduling heuristics are necessary to explore "interesting" bytecode programs first, instead of naïvely checking all possible bytecode programs. Furthermore, to symbolically execute JITted bytecode, one needs to model the symbolic engine at the native machine instruction level. An alternative may be to transform the machine code back to some intermediate representation, and then prove its equivalence to the original bytecode.

Build extensible interpreters. Since it is error-prone to build an interpreter from scratch, one strategy is to reuse sound bytecode formats and construct an extensible interpreter, which consists of reusable components, similar to the Xoc extensible compiler [5]. Developers can build and customize an interpreter by choosing a set of components, such as whether the interpreter supports integer arithmetic, or whether it allows loops. For example, Seccomp [7], a security mechanism for isolating processes, reuses the existing BPF interpreter to run BPF bytecode for specifying system call filtering rules, instead of implementing an interpreter from scratch. One research question is whether a single extensible interpreter can address sometimes-conflicting requirements of different use cases, such as minimal runtime complexity, high throughput, low latency for short bytecode, performance isolation, and support for general-purpose C-like programs.

6 Conclusion

In this paper, we analyzed security bugs found in embedded interpreters, classified their effects, and suggested ways to reduce common vulnerabilities. We hope our results can shed some light on how to design secure embedded interpreters in the future.

Acknowledgments

Thanks to the anonymous reviewers for their feedback. This work was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] Bitcoin. Script - Bitcoin, 2013. https://en.bitcoin.it/wiki/Script.

[2] D. Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. *BlackHat DC*, 2010.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[4] J. Corbet. A JIT for packet filters, Apr. 2011. http://lwn.net/Articles/437981/.

[5] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 244–254, Seattle, WA, Mar. 2008.

[6] DragonFlyBSD. Dragonflybsd bug 1748, 2010. http://bugs.dragonflybsd.org/issues/1748.

[7] W. Drewry. SECure COMPuting with filters, Jan. 2012. http://lwn.net/Articles/498231/.

[8] D. R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–170, Philadelphia, PA, June 1996.

[9] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):41–44, Jan. 2012.

[11] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification, Dec. 2011. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf.

[12] A. S. Incorporated. The type 2 charstring format, Mar. 2000. http://partners.adobe.com/public/developer/en/font/5177.Type2.pdf.

[13] A. Kuznetosv. SS utility: Quick intro, Sept. 2001. http://www.cyberciti.biz/files/ss.html.

[14] K. McAllister. Attacking hardened Linux systems with kernel JIT spraying, Nov. 2012. http://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

[15] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Annual Technical Conference*, San Diego, CA, Jan. 1993.

[16] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. http://bitcoin.org/bitcoin.pdf.

[17] T. Ormandy. Fun with constrained programming, 2012. http://blog.cmpxchg8b.com/2012/09/fun-with-constrained-programming.html.

[18] A. Pitrou. Pickle protocol version 4. PEP 3154, Dec. 2011. http://www.python.org/dev/peps/pep-3154.

[19] R. Price, C. Bormann, J. Christoffersson, H. Hannu, Z. Liu, and J. Rosenberg. Signaling compression (SigComp). RFC 3320, Jan. 2003. http://www.ietf.org/rfc/rfc3320.txt.

[20] C. Reis, A. Barth, and C. Pizano. Browser security: Lessons from Google Chrome. *Communications of the ACM*, 52(8):45–49, June 2009.

[21] C. Rohlf and Y. Ivnitskiy. The security challenges of client-side just-in-time engines. *Security & Privacy, IEEE*, 10 (2):84–86, 2012.

[22] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, Alexandria, VA, Oct.–Nov. 2007.

[23] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, Washington, DC, Oct. 2004.

[24] J. Sigwald. Analysis of the jailbreakme v3 font exploit, July 2011. http://esec-lab.sogeti.com/post/Analysis-of-the-jailbreakme-v3-font-exploit.

[25] M. Slaviero. Sour pickles: Shellcoding in Python's serialisation format, 2011. https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_WP.pdf.

[26] TrueType. The instruction set, 2011. https://developer.apple.com/fonts/TTRefMan/RM05/Chap5.html.

[27] TrueType. TrueType hinting, 2012. http://www.truetype-typography.com/tthints.htm.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, Dec. 1993.

[29] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–177, Hollywood, CA, Oct. 2012.

[30] A. Wu. Bytecode signatures for polymorphic malware, Nov. 2010. http://blog.clamav.net/2011/11/bytecode-signatures-for-polymorphic.html.