Enabling Sophisticated Analyses of x86 Binaries with RevGen

Vitaly Chipounov and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Current state-of-the-art static analysis tools for binary software operate on ad-hoc intermediate representations (IR) of the machine code. Therefore, even though IRs facilitate program analysis by abstracting away the source language, it is hard to reuse existing implementations of analysis tools in new endeavors. Recently, a new compiler framework — LLVM— has emerged, together with many analysis tools that use its IR. However, these tools rely on a compiler to generate the IR from source code.

We propose RevGen, a tool that automatically converts existing binary programs to the standard LLVM IR, making an increasingly large number of static and dynamic analysis frameworks, as well as run-time instrumentation tools, applicable to legacy software. We show the potential of RevGen by converting several programs and device drivers to LLVM and checking the resulting code with off-the-shelf analysis tools.

1 Introduction

There exist many powerful tools for various types of code analysis. For example, BitBlaze [25] combines dynamic and static analysis components to extract information from malware. CodeSurfer [2] can perform program slicing, to allow understanding code behavior. Calysto [1] is a static bug finder and `bddbddb` [19] provides a framework for querying programs for buggy code patterns.

Unfortunately, most of these tools require source code. Coverity [6], `bddbddb`, Saturn [14], and various abstract-based representation methods [7] require C code. Other tools like Java PathFinder [23] or CoreDet [5] rely on a Java and LLVM compilers to transform the source code to their analysis format.

The reliance on source code leaves a significant portion of legacy and proprietary software unanalyzed. Even when the source is partially available, parsing it can be challenging [6] and the presence of binary libraries or even inline assembly can severely degrade the performance of both static and dynamic analysis tools. Bug finding and debugging tools like KLEE [9] and ESD [29] cannot work on such programs.

There also exist tools that directly analyze machine code, but they often use ad-hoc intermediate representations (IR), making it hard to extend them to other architectures and preventing easy reuse of analysis components. An IR abstracts the source language (e.g., C or assembly) to facilitate analysis. For example, CodeSurfer is based on the IR generated by the proprietary IDAPro [16] disassembler, while Jakstab [17] relies on the frontend of the Boomerang [8] decompiler, and Vine, the static analysis component of BitBlaze, uses yet another representation.

In recent years, LLVM gained a large popularity, becoming a platform of choice for developing new source-based analysis tools, and arguably imposing its IR as a *de facto* standard for such tools. Currently, more than 160 LLVM-based projects are developed [20], with numerous static analysis tools targeted at software verification [24, 12, 3, 28, 24], as well as instrumentation tools enforcing safety properties at run-time, like deterministic execution [5], dynamic bug finders [18, 1], or safe execution of error recovery code [15]. LLVM is now actively supported by Apple and forms the basis of several commercial applications, e.g., MacOS and Xcode.

Several powerful analysis frameworks have been built with LLVM. KLEE looks for bugs in programs using symbolic execution, a method for thorough path exploration. KLEE found deep bugs in Coreutils that were overlooked for a decade. Parfait [12] is an LLVM-based static analysis framework that scales to millions of lines of code using demand-driven analysis. Finally, LLBMC [24] is a tool that applies bounded model checking to LLVM programs.

In this paper, we present RevGen, a tool that enables the reuse of LLVM-based analysis frameworks on legacy binary software. RevGen uses static binary translation to convert binary code to the widely-used LLVM IR, without relying on the source code. The output of the tool is an LLVM program that can be analyzed, instrumented, and executed by standard, off-the-shelf, LLVM-based analysis frameworks.

In the rest of the paper, we show examples of use cases that RevGen enables (§2), the challenges RevGen faces (§3), present the design and implementation (§4), expose preliminary results (§5), discuss (§6), and conclude (§7).

2 Use Cases

In this section, we illustrate how RevGen can be used in practice with existing analysis tools that are based on LLVM and that implicitly rely on the availability of the source code. We show the use cases of deterministic program execution, bug finding in kernel-mode binaries using static analysis, reverse engineering of device drivers for safety and portability, inline assembly removal, and analysis of embedded software.

Debugging multi-threaded programs Multi-threaded programs are particularly prone to bugs. Threads share data and use synchronization mechanisms, which can potentially lead to data races and deadlocks. The difficulty of debugging these problems is compounded by the presence of synchronizations implemented in an ad-hoc way [27]. Tools like CoreDet [5] and SyncFinder [27] make debugging of concurrency bugs easier. However, they only run on LLVM code.

RevGen allows SyncFinder to annotate blocks of binary code that use ad-hoc synchronization. SyncFinder locates the loops in the LLVM code, analyzes exit conditions, determines which blocks of code can run concurrently, and whether the exit condition can be affected by concurrent writes. If it is the case, SyncFinder reports an ad-hoc synchronization.

Likewise, RevGen enables the use of CoreDet on binary programs. CoreDet is a compiler and runtime environment that instruments multi-threaded programs in order to make them behave deterministically. CoreDet ensures that all conflicting concurrent stores are performed in a specific sequence and that threads are created and scheduled in a fixed order, while introducing as little serialization as possible.

Analyzing kernel-mode code Proprietary binary drivers are a major source of system crashes and unreliability. On Linux, error rate in drivers is 3-7 times higher than in the rest of the kernel [11]. Windows drivers are no better, causing 85% of crashes [22]. Since drivers usually run in kernel mode at the highest privilege level, exploiting their bugs can lead to complete denial of service and full system compromise.

By converting binary drivers to LLVM, RevGen would enable the use of static analysis tools on such drivers. LLBMC [24] is a static analysis tool that checks properties like integer overflows, illegal memory accesses, buffer overflows, or invalid bit shifts. Its abilities make it one of the first choices to verify device drivers.

RevGen also enables static analysis of low-level OS code. Such code typically uses machine instructions that have no equivalent in programs written in high-level languages. The challenge is to accurately emulate these instructions using the LLVM IR in order to make them amenable to static analysis.

Reverse engineering safe drivers Static analysis tools are useful to check the quality of drivers but they cannot fix buggy drivers by themselves. Moreover, such tools are of little help to users who are often forced to load faulty drivers because there is no better choice. Even if run-time driver bug containment tools exist [26], they incur overhead and are limited to a few OSes. Ideally, there should be a tool that automatically fixes buggy drivers.

RevNIC [10] uses reverse engineering to synthesize safer drivers from buggy ones. RevNIC takes a binary driver and traces its execution to observe all the ways in which the driver interacts with the hardware. The traces contain LLVM instructions complemented with dynamic I/O, memory, and register data, that RevNIC uses to encode the hardware-interaction state machine.

RevGen can be used to improve the synthesized drivers. RevNIC has low code coverage on complex device drivers, resulting in incomplete LLVM code and reduced driver functionality, which forces users to manually write the missing code. Even though RevGen cannot recover trace data, it can automatically transform the missing code to LLVM, minimizing manual intervention.

Helping source-based tools LLVM supports native inline assembly, whose presence prevents most of the state-of-the-art analysis tools from running properly. To analyze such functions accurately, analysis tools must precisely model the semantics of each machine instruction (i.e., what the instruction does). Failing to do so may cause both false negatives and false positives. For example, KLEE [9] aborts execution paths that have inline assembly and static analysis tools either ignore or make unsound assumptions about such code [6].

Inline assembly is common in large applications. For instance, network applications use byte-order conversion routines (e.g., `htons`) implemented with specific machine instructions, while multimedia libraries use inline assembly to efficiently implement various algorithms.

While such code can be tedious to transform to C by hand, RevGen can do it automatically. RevGen scans the LLVM code, extracts inline assembly, identifies input/output parameters, wraps the assembly into separate LLVM functions, and uses `llvm-gcc` to turn these functions into binary code. Finally, RevGen translates the obtained binary code back to pure LLVM, which it uses as a drop-in replacement of the inline assembly.

Analyzing embedded software While x86 is a common architecture on desktop PCs and servers, there are many more architectures in the embedded world. For instance, smartphones use MIPS and ARM processors. RevGen can automatically convert instruction sets of these platforms to LLVM. This immediately allows the reuse of LLVM-based tools on embedded proprietary software. We shall see in the next section how RevGen's design enables the support of different architectures.

3 Challenges

Enabling static analysis of machine code poses two main challenges for static translators like RevGen: extracting binary code's semantics and inferring type information.

First, translators must extract the semantics of the machine instructions. For this, they decompose each complex instruction in a sequence of simpler operations (the intermediate representation). However, virtually all tools ignore the system instructions that manipulate the control state (e.g., switching execution modes, loading segment registers on x86, etc.). Therefore, such tools cannot analyze OS kernel code accurately. Finding bugs such as privilege escalation through virtual 8086 mode (affecting all Windows versions from NT 3.1 to Windows 7 [21]) is out of reach for them. RevGen addresses this challenge.

Second, translators must infer type information to enable accurate analysis. The LLVM IR is designed to retain most of the type information present in the source code. However, binaries only manipulate integers and memory addresses. The absence of type information degrades the quality of some analyses, in particular alias analysis. Analyses that rely on precise alias information have their rate of false positives and negatives increased.

The challenge for RevGen is to rebuild the type information and other LLVM constructs as if the resulting LLVM code was obtained by compiling source code. This places RevGen in between disassemblers and decompilers. While disassemblers stop after generating the IR, decompilers turn the IR into human-readable high-level code, after reconstructing type information, variables, control flow, etc. RevGen does not need to reconstruct high-level control flow.

4 RevGen Prototype

RevGen takes as input an x86 binary and outputs an equivalent LLVM module in three steps. The general architecture is shown in Figure 1. First, RevGen looks for all executable blocks of code and converts them to LLVM translation blocks (§4.2). Second, when there are no more translation blocks (TB) to cover, RevGen transforms them into basic blocks and rebuilds the control flow graph (CFG) of the original binary in LLVM format (§4.3). Third, RevGen resolves external function calls to build the final LLVM module. For dynamic analysis, a last step links the LLVM module with a run-time library that allows the execution of the LLVM module (§4.4).

4.1 Background

LLVM is a compiler framework that uses a compact RISC-like instruction set with an unlimited number of registers. LLVM has about 30 opcodes, only two of which can access memory (`load` and `store`), all other instructions operate on virtual registers.

LLVM uses the static single assignment (SSA) code representation. In SSA, each register can be assigned only once. Hence, SSA also provides a `phi` instruction that assigns values to variables depending on the direction of the control flow. This instruction allows to modify the same variable in two different branches.

This makes LLVM programs amenable to complex analyses and transformations. LLVM code explicitly embeds the program's data flow and def-use graphs. This enables transformations like function inlining, constant propagation, or dead store removal, which are a key part of static and dynamic analysis tools.

A static translator must take into account LLVM specificities. It must account for pointer arithmetic, accommodate different stack layouts, transform accesses to various code and data segments, deal with indirect calls, and provide runtime support to be able to execute the generated LLVM programs. Finally, the translated code must be semantically-equivalent to the original binary.

4.2 Translating Blocks of Binary Code

The static translator takes as input the binary file and a program counter and transforms all the machine instructions to LLVM until it encounters a terminator. A terminator is an instruction that modifies the control flow (e.g., branch, call, return). The translation has two steps: the input is first disassembled into micro-operations, which are then converted to LLVM instructions.

First, the translator converts machine instructions into an equivalent sequence of micro-operations. For example, the x86 instruction `inc [eax]` that increments the memory location pointed to by the `eax` register is split into a load to a temporary register, an increment of that register, and a memory store. The sequence of micro-operations forms a *translation block*.

Second, the translator maps each micro-operation to LLVM instructions, using a code dictionary. The dictionary associates each micro-operation with a sequence of LLVM instructions that implement the operation. Most conversions are a one-to-one mapping between the micro-operations and the LLVM instructions (e.g., arithmetic, shift, load/store operations).

The translator also takes into account instructions that manipulate the system state. Current tools do not model such instructions to a sufficient precision level. For example, RevGen accurately translates to LLVM instructions like `fsave` or `mov cr0, eax`. The former saves the state of the floating point unit, while the latter sets the control register (e.g., to enable 32-bit protected mode, which changes the behavior of many instructions).

For this, the translator uses *emulation helpers*. An emulation helper is a piece of C code that emulates complex machine instructions that do not have equivalent micro-operations. RevGen compiles emulation helpers to LLVM and adds them to the code dictionary, transparently enabling the support of machine instructions that manipulate system state.

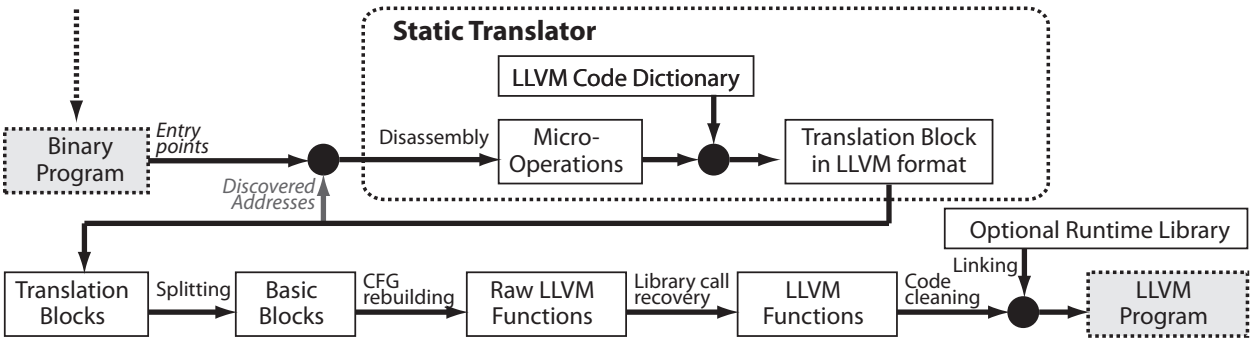

Figure 1: The RevGen Workflow

Third, the translator packages the sequence of LLVM instructions into an LLVM function that is *equivalent* to the original binary code. More precisely, given the same register and memory input, the translated code produces the same output as what the original binary does if executed on a real processor.

The translator stops when all translation blocks have been extracted. This happens when the translator cannot find new code to disassemble (e.g., by looking at not-yet explored jump and call target addresses).

4.3 Reconstructing the Control Flow Graph (CFG)

The CFG builder converts the translation blocks to basic blocks and groups the basic blocks into LLVM functions. The resulting functions are equivalent to those implemented by the original binary program.

RevGen generates basic blocks by splitting translation blocks whenever necessary. A basic block is a sequence of instructions that has only one entry and one exit point. Unlike translation blocks, no code can jump to the middle of a basic block. When this happens, RevGen splits the block at the target instruction, yielding two different LLVM functions. This happens iteratively until no more splitting is possible (i.e., only basic blocks remain).

Next, RevGen identifies the function entry points. RevGen considers basic blocks that are targets of `call` instructions or that have no incoming edges to be function entry points. We found this heuristic to work well on a variety of binaries produced by standard compilers.

RevGen builds the CFG of each function by connecting basic blocks together. Two basic blocks are connected if they follow each other or if the second basic block is the target of a jump in the first one.

Finally, RevGen transforms the CFG into an LLVM function. RevGen represents each basic block b of the original binary by an LLVM function f_b. RevGen first inserts an LLVM `call` instruction to the next basic block at the end of each f_b. Then, RevGen applies an LLVM function inlining pass to merge all the call targets into one large LLVM function.

4.4 Obtaining Analyzable LLVM Programs

The output of the CFG builder is a *raw* LLVM function that cannot be used by static or dynamic analyzers as is (e.g., it lacks explicit library calls and contains unresolved pointer arithmetic). We describe next how to transform the CFG builder output into analyzable code.

RevGen makes several assumptions about the original binary to synthesize analyzable LLVM code. RevGen requires the binary to provide a symbol table to identify library calls and a relocation table to identify all constants used as pointers. Moreover, the binary must not have self-modifying code. Finally, both the source and target architectures must have the same pointer size, in order to be able to run the translated code.

4.4.1 Enabling Static Analysis

First, RevGen identifies external function calls by scanning the import table of the program binary. An import table maps a list of function and library names to addresses. The OS loader patches the table with the actual function adresses so that indirect calls that reference the table can work properly.

Second, RevGen patches the raw LLVM functions with explicit external calls. Basic blocks originally encode external calls by an indirect jump to an address read from the import table. RevGen replaces such jumps by LLVM `call` instructions using the actual function names, allowing the LLVM linker to later resolve the call targets. This step is required because static analysis tools look for the use of specific functions. For example, memory checkers would track the calls to `malloc` and `free`.

Third, RevGen encodes the content of the program's segments as LLVM arrays and embeds them in the LLVM program. This preserves the assumptions about the data layout in the original binary and account for programs that refer to segments with pointer arithmetic.

RevGen does not need to resolve indirect control flow (ICF). RevGen is not a static analysis tool, it only translates ICF from x86 to LLVM. Static resolution of ICF is left to the analysis tools. Such tools can resolve ICF to any precision they need and provide any soundness and completeness guarantees they wish.

4.4.2 Enabling Dynamic Analysis

Dynamic tools perform analysis at run-time, which requires to execute the program. For this, besides applying the steps described in §4.4.1, RevGen links a specific runtime library that resolves indirect function calls, deals with memory layouts, and handles multi-threading. This library does not affect run-time analysis tools, because they have full access to the library's *bitcode* (i.e., the binary representation of the LLVM IR). They see the runtime as another component of the analyzed program.

Resolving pointer arithmetic RevGen uses relocation tables to identify all pointers in the binary and adapt them to the LLVM memory model. Such tables list all code and data locations that the OS loader patches if it loads the binary at a different base address than what the compiler assumed. RevGen uses these tables to translate all hard-coded pointers to LLVM pointers, adapted to the memory layout seen by LLVM. In particular, RevGen remaps pointers that reference data segments to the corresponding LLVM arrays. Without relocation tables, RevGen resorts to monitoring memory accesses and patching them at run-time.

Resolving indirect calls and branches The code generator embeds a table that maps basic blocks' native addresses to the corresponding LLVM basic blocks. It also stores which function the basic blocks belong to, as well as whether the basic block is the entry point of a function.

Whenever the translated code performs an indirect call or jump to a native address, the runtime looks for the corresponding LLVM basic block. If the block is not found or if there is a type mismatch (e.g., calling a block that is not a function entry point), the runtime aborts the program and notifies the user.

Adapting stack pointers The translated code retains all the assumptions of the original binary about the stack layout. In particular, it assumes that local variables, return addresses, and parameters are located at precise memory adresses when doing stack pointer arithmetic.

The runtime library preserves the original stack layout by using a dual-stack architecture. There is one *native* stack used by the LLVM program and one *implicit* stack, whose pointer is passed as a parameter to each LLVM function, and which is manipulated by the LLVM functions. The runtime allocates the implicit stack and sets the implicit stack pointer before calling the main entry point of the program. It also copies the arguments to the native stack when calling library functions.

Supporting multi-threading Multi-threading consists in allocating an implicit stack for each thread. For this, the runtime library intercepts thread allocation routines and wraps the thread entry points into a special function that sets up the implicit stack. The stack is automatically freed when the thread routine finishes.

Self-modifying code RevGen does not support self-modifying code. This does not hurt RevGen because such code is practically restricted to JITed languages (e.g., C#, Java). In these cases, existing tools run on the bytecode of the respective languages, not on the final machine code itself. A side effect of this lack of support is that native code injected at run-time cannot be executed, increasing the safety of translated programs, similar to what other LLVM-based tools aim at achieving [13].

5 Preliminary Results

In this section, we aim to answer two key questions: Does RevGen enable the reuse of existing LLVM-based analysis tools on x86 binaries? What completeness can RevGen achieve on typical binaries?

We evaluate a prototype of RevGen that is based on the QEMU's [4] binary translator. QEMU is a system emulator that runs unmodified OSes on top of arbitrary hosts by dynamically translating the guest instructions to the host's instruction set. We extended the dynamic translator to generate LLVM bitcode in prior work [18, 10]. In this work, we separate the dynamic translator from QEMU and make it static.

To answer the first question, we convert an x86 micro-benchmark to LLVM using RevGen and run the result in CoreDet [5] . The micro-benchmark has several threads that access unprotected shared variables, whose value is printed at the end of each run. Without CoreDet, the printed output differs from run to run. With CoreDet, the output stays the same. This shows that RevGen enabled the reuse of CoreDet to render binary programs behave deterministically in the presence of race conditions.

Initial results suggest that RevGen's completeness is comparable to state-of-the-art disassemblers on kernel-mode binaries. We disassembled the `pcntpci5.sys` Windows network device driver with RevGen and compared the results to IDA Pro. IDA Pro identified 78 functions, while RevGen found 77. RevGen failed to find 4 functions and misinterpreted 3 basic blocks as function starts, because of incomplete detection of jump tables.

6 Discussion

In this section, we discuss three aspects that we believe will enable RevGen to become a major enabler for widespread static analysis of binary programs.

RevGen can effortlessly leverage existing disassemblers, should better completeness be required. RevGen's translator only requires a list of program counters and an accurate list of function entry points in order to convert the binary to LLVM. Both can be directly obtained from disassemblers like IDAPro or state-of-the-art static analyzers such as Jakstab.

Extending RevGen to support other architectures than x86 is simple and requires limited efforts. The LLVM backend that translates micro-operations to LLVM need

not be modified. The only need is to modify the frontend (e.g., the ARM or MIPS frontend) with annotations specifying the types of basic blocks (e.g., branch, call, return, etc.) to allow the CFG builder to merge the basic blocks and reconstruct the functions.

We argue that RevGen enables analysis tools to check binary programs as well as their interaction with the processor. Analysis tools typically check programs that interact with libraries. In the context of RevGen, the program is the machine code translated to LLVM and the library is the collection of emulation helpers in LLVM format. For example, checking that the invocation of a software interrupt does not cause a general protection fault is reduced to verifying that the library does not invoke the corresponding program's entry point.

This can potentially open up all sorts of analyses on low level system code. We envision RevGen to enable analysis tools to answer questions like: Can the user-mode code issue a system call in such a way that would cause arbitrary code execution? Are there any bugs in the emulation helpers (and thus in QEMU) that would cause the application to malfunction? Since RevGen produces plain LLVM bitcode, we expect existing tools to answer such questions out of the box.

7 Conclusion

We presented RevGen, a tool that automatically converts existing binary programs to the LLVM intermediate representation. RevGen can potentially enable a large number of static and dynamic analysis frameworks, as well as run-time instrumentation tools to work on legacy software. Preliminary results show that RevGen can successfully translate large Windows drivers and run existing dynamic analysis tools on binary programs. We plan to make the RevGen prototype freely available.

References

[1] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. 2008.

[2] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. Technical report, 2005.

[3] B. Bartels and S. Glesner. Formal modeling and verification of low-level software programs. In *Intl. Conf. on Quality Software*, 2010.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 2010.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Conf. on Programming Language Design and Implem.*, 2003.

[8] Boomerang decompiler. http://boomerang.sourceforge.net/.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[10] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

[11] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. 2001.

[12] C. Cifuentes and B. Scholz. Parfait — designing a scalable bug checker. In *Static Analysis Workshop*, 2008.

[13] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Symp. on Operating Systems Principles*, 2007.

[14] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Conf. on Programming Language Design and Implem.*, 2008.

[15] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. We crashed, now what? In *Workshop on Hot Topics in Dependable Systems*, 2010.

[16] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com.

[17] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *10th Intl. Conf. on Formal Methods in Computer-Aided Design*, 2010.

[18] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conf.*, 2010.

[19] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Symp. on Principles of Database Systems*, 2005.

[20] C. Lattner. LLVM related publications. Official LLVM web site. Retrieved on 2010-12-04. http://llvm.org.

[21] MITRE. CVE-2010-0232: Microsoft Windows NT #GP trap handler allows users to switch kernel stack, 2010. http://cve.mitre.org.

[22] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft Windows Hardware Engineering Conf., 2003.

[23] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[24] C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *5th Intl. Workshop on Systems Software Verification*, 2010.

[25] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Intl. Conf. on Information Systems Security*, 2008.

[26] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), 2005.

[27] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad-hoc synchronization considered harmful. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[28] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In *Intl. SPIN Workshop*, 2008.

[29] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.