BinRec: Attack Surface Reduction Through Dynamic Binary Recovery

Taddeus Kroes
Vrije Universiteit Amsterdam

Anil Altinay
University of California, Irvine

Joseph Nash
University of California, Irvine

Yeoul Na
University of California, Irvine

Stijn Volckaert
University of California, Irvine

Herbert Bos
Vrije Universiteit Amsterdam

Michael Franz
University of California, Irvine

Cristiano Giuffrida
Vrije Universiteit Amsterdam

ABSTRACT

Compile-time specialization and feature pruning through static binary rewriting have been proposed repeatedly as techniques for reducing the attack surface of large programs, and for minimizing the trusted computing base. We propose a new approach to attack surface reduction: dynamic binary lifting and recompilation.

We present BinRec, a binary recompilation framework that lifts binaries to a compiler-level intermediate representation (IR) to allow complex transformations on the captured code. After transformation, BinRec lowers the IR back to a "recovered" binary, which is semantically equivalent to the input binary, but has its unnecessary features removed. Unlike existing approaches, which are mostly based on static analysis and rewriting, our framework analyzes and lifts binaries dynamically. The crucial advantage is that we can not only observe the full program including all of its dependencies, but we can also determine which program features the end-user actually uses. We evaluate the correctness and performance of BinRec, and show that our approach enables aggressive pruning of unwanted features in COTS binaries.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Software reverse engineering**;

KEYWORDS

Binary lifting; attack surface reduction; symbolic execution; LLVM

ACM Reference Format:
Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. 2018. BinRec:, Attack Surface Reduction Through Dynamic Binary Recovery. In *The 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3273045.3273050

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FEAST '18, October 19, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5997-9/18/10…$15.00
https://doi.org/10.1145/3273045.3273050

1 INTRODUCTION

As a software program evolves over time, developers often introduce additional features to address various user expectations and to improve compatibility with other software or hardware. Many of these additional features remain unused by regular users, however, and security vulnerabilities in them often remain undiscovered for years. In a recent study, Wagner et al. found that 83% of security vulnerabilities that were assigned a CVE number in 2014 laid in âĂIJcold codeâĂİ [33]. To discover bugs lurking in cold code, one current practice is to insert sanity checks into a program using dynamic bug detection tools (e.g., sanitizers), and to fuzz the program or run test cases to exercise all parts of the program. As program complexity grows, these approaches are less suitable for detecting bugs hidden in deep execution paths.

An alternative approach to eliminate latent bugs is to specialize the program for specific use cases through manual feature pruning [24], compile-time specialization [22], or link-time code compaction [9]. Assisted by either static or dynamic analysis to determine which features are unnecessary for the target use case, these techniques can achieve substantial reductions of the program's code size, while also removing the features that are the most likely to contain security vulnerabilities. The downside of these techniques is that they are heavily tailored to the kernel, exploiting the fact that the kernel has a small set of easily recognizable interfaces to external code (i.e., user-space programs, peripheral devices, etc.), and thousands of preprocessor options to enable or disable features at the source code level.

We propose a more generic approach for attack surface reduction that works for binary commercial off-the-shelf (COTS) programs. Instead of trying to find vulnerabilities in cold code, or running coarse-grained analyses to identify unnecessary features and remove them at compile time, we use fine-grained dynamic profiling to determine with greater precision which parts of a program's code are actually used. We then lift these parts of the code into a compiler intermediate representation (IR) format, and use a mainstream compiler to compile the IR to a new binary executable.

We also present BinRec, a framework that implements our proposed approach. BinRec is a binary program recovery and recompilation framework that combines dynamic profiling with multipath exploration to lift machine instructions to compiler IR. It then transforms the IR using compiler passes (e.g., to optimize and harden the recovered program), and recompiles it to executable code. To reduce the attack surface exposed by unwanted program features,

we selectively recover only the essential parts of the program based on execution tracing and symbolic execution. Preliminary results show that BinRec effectively reduces the attack surface, and enables complex transformations using existing compiler passes.

To summarize, our contributions are threefold:

- We present a novel approach to attack surface reduction through binary recovery, combining selective binary lifting with program analysis and transformation at the level of compiler IR, based on dynamic profiling.
- We implement our approach in BinRec and apply our framework to real-world COTS binaries.
- Our results show that BinRec successfully reduces the attack surface in the recovered binary by removing unused program paths and by reducing the number of ROP gadgets.

2 BACKGROUND

There are several existing efforts attempting binary recovery or recompilation, but their specific aims vary. Some solutions provide static reassembleable disassembly [34], but they are not conservative and, without lifting to IR, they do not provide access to powerful compiler-level analysis. Revgen [11] and Dagger [5] can disassemble binaries to LLVM bitcode, but the IR they produce contains machine specific constructs and unresolved control flow. It would require extensive tool-specific analyses and transformations before that IR could be coherently understood by off-the-shelf compiler analyses. SecondWrite [3] and McSema [16] aim to lift entire binaries to LLVM IR and recompile them. Rev.ng [15] can lift entire binaries to compilable LLVM IR, and primarily aims to provide analysis of programs' control flow and function boundries. However, these approaches are all fully *static*: static disassembly, static analysis, and static rewriting. Static disassembly decodes machine instructions in a binary without executing them. When successful, this provides the advantage of a complete program view. Unfortunately, static analysis is difficult for complicated code. SecondWrite [3], for instance, cannot correctly handle handwritten assembly, special idioms, many optimizations and transformations, or even position-independent code. Still, it is sufficient for benign off-the-shelf binaries that follow known code generation patterns from compilers and contain symbols. Unfortunately, binaries that are the target of analysis are typically release builds, stripped of symbols and debug information. In addition, such static disassembly tools conservatively assume that all statically reachable program paths are equally important in the absence of run-time information. Such conditions make static disassembly ineffective for attack surface reduction through binary rewriting. Hence, we propose to use *dynamic* analysis to support cases where static disassembly inherently fails, instead following actual runs of the program. While this removes the problems of static analysis, it introduces new ones. For instance, what should be done when a rewritten binary hits an execution path that did not appear during profiling and how to improve coverage to limit such cases?

3 BINREC DESIGN

Figure 1 shows an overview of the BinRec framework, consisting of three components: a front-end that lifts machine code to compiler IR, a transformation component that rewrites the IR in any way

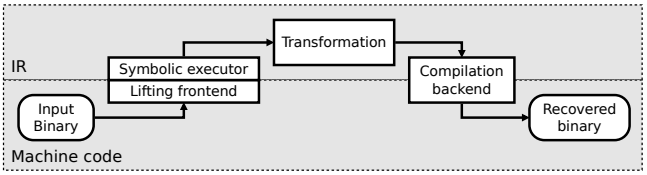

Figure 1: The steps of binary recovery: lifting to compiler IR, transformation on the IR level, and lowering of compiled IR back to machine code.

desired (e.g., compiler optimizations, security hardening, etc.), and a back-end that compiles the transformed IR back to machine code and produces an executable binary. At a high level, BinRec works as follows. First, the lifting front-end executes the given program binary with concrete inputs in a virtual environment. At the same time, it also selectively invokes symbolic execution to discover possible execution paths that may be reachable with other legitimate program inputs. The front-end lifts all the program paths visited during both the concrete and symbolic execution, and turns them into IR snippets. It then removes any dependency with the virtual environment from the lifted IR and merges them into a single IR module that can be analyzed by the subsequent transformation passes. The transformation component can incorporate any technique that operates on compiler IR, including existing optimization and security hardening passes. Finally, the back-end component removes any remaining instrumentation added by the front-end, and compiles the final IR to machine code using the compiler.

Unlike traditional binary recovery tools, BinRec does not aim to maximize code coverage. Instead, it aims to reduce the program's attack surface by removing rarely executed code paths. With this strategy, however, the recovered binary may not include all the necessary program functionality, especially when the recovered binary is later executed with untested program inputs. To guarantee that the recovered binary will execute correctly, we enforce a fallback to the original code whenever it hits execution paths removed from the recovered binary. Thanks to our trace-based lifting, fallbacks should not occur frequently. We can therefore afford to embed heavyweight (i.e., slow) security mechanisms in the fallback code, which allows us to further reduce the program's attack surface.

3.1 Lifting binary code to compiler IR

We discover additional possible execution paths by performing multi-path recovery of the input binary. To do so, we employ symbolic execution [8, 21], a well-known technique to find software bugs by replacing memory values with symbolic expressions. To ensure conservativeness, we must handle all execution paths that both the profile runs and symbolic execution missed in a conservative manner. In other words, the binary should detect a deviation from the observed code paths and take the appropriate action (e.g., jump to to existing machine code in the input binary).

We base our design on a lifting front-end that lifts machine code to IR using a virtual machine for program isolation. The front-end produces IR that is specific to the execution environment. For instance, both data and code pointers are put as constants in the IR. This also holds true for position independent code (PIC), which

normally uses only offsets relative to the location of the program. This means that our design supports the lifting of PIC binaries by fixing the code and data locations that were used during tracing, and thereafter accessing them in an address dependent manner.

3.2 Ensuring conservative program behavior

While trace-based lifting of BinRec is mainly designed for attack surface reduction, it can also be used for aggressive input-specific optimizations. BinRec features a configurable fallback mechanism that facilitates both use cases. With the fallback enabled, the recovered program diverts any control flow edge that ventures outside of the recovered CFG to the corresponding machine code in the input binary. The fallback mechanism currently supports four modes:

(1) **disabled** No fallback, inputs that require control flow transfers outside the recovered path trigger undefined behavior. This allows for aggressive optimization, but does not preserve program behavior for all inputs.
(2) **error** Unknown control flow transfers trigger an error. This is used when a fallback is not desired (e.g., because the code is only considered secure in the instrumented path), but the input cannot be controlled to always be confined to the recovered path (which the **disabled** mode requires).
(3) **standard** Any unknown control transfer jumps to the corresponding instruction in the existing code of the input binary. The binary maintains a jump table of code pointers to all recovered basic blocks. The target of the control flow transfer is first checked to exist in the jump table—to stay in the recovered path for as long as possible—and execution moves to existing code only in case of a miss.
(4) **replaced** Based on standard mode, but we also modify the existing code in the input binary to jump to recovered code for a selected set of code addresses (which defaults to all recovered code addresses). This enforces execution of lifted code for the selected addresses, while conservatively maintaining all original behavior of the binary. **replaced** mode is not yet implemented in our current prototype.

Intuitively, the attack surface is enlarged when adding code to a binary while also leaving the original code in place, as is done by the **standard** fallback mode. Depending on the type of attack surface reduction desired, however, the original code can be modified to mitigate attacks. For instance, when implementing memory safety or control flow integrity, the original code can be monitored by a heavyweight binary instrumentation framework. The recovered code constitutes a fast path in this case in which instrumentation is efficiently embedded within the recovered IR before recompilation. When reducing the amount of ROP gadgets, on the other hand, one would use the **error** mode and remove the original code sections.

3.3 Deinstrumentation

Running a binary in a virtual machine requires maintaining a virtual processor state (e.g., registers) in memory, which is also visible in the lifted code that interacts with this virtual environment rather than on physical registers. A significant part of our work concerns *deinstrumenting* the captured code—detaching the captured IR from the runtime environment in order to create a standalone executable binary. We omit details of these transformations here due to space

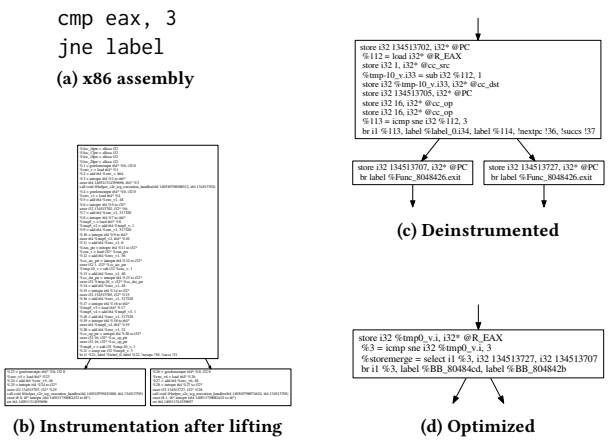

Figure 2: Deinstrumentation of a small basic block (a). Dynamic code lifting captures instrumented, decoupled code (b). Deinstrumentation shortens the code and adds explicit control flow instructions (c). After applying standard compiler optimizations, a single basic block remains in the IR (d). Note that the instructions illustrate code size and are not intended to be readable.

constraints, but Figure 2 shows their efficacy on a small code example. Conversely, we also instrument the IR with some additional information, such as memory accesses, to aid in performing effective optimizations. The added instrumentation is removed before compiling the IR back to executable code.

3.4 Producing an executable binary

The back-end component removes any remaining instrumentation added by the front-end, and compiles the rewritten IR to machine code. Although lifted instructions are ready for compilation to standalone machine code, dependencies on data and external functions from the original program are still unresolved in the resulting binary. Uroberos [34] attempts to find and replace such references in the code with position-independent references in the output binary. They rely on the assumption that all such references can be found, which is not guaranteed in all use cases we aim to support. Instead, we follow the more conservative approach of SecondWrite [3], which embeds sections from the input binary in the recovered IR and enforces linkage at the original base addresses.

4 IMPLEMENTATION

We have implemented our approach in BinRec, spanning 9960 sloc of which 7915 are C++ code that implements lifting and deinstrumentation. The implementation targets 32-bit x86 binaries on Linux.

BinRec uses S^{2}E [12] as its lifting front-end. S^{2}E implements *selective symbolic execution*, alternating computationally intensive symbolic execution with high performance dynamic binary translation (DBT). This facilitates symbolic execution of a single process running in a QEMU [4] virtual machine. Code is translated to LLVM IR in order to be symbolically executed by the KLEE [7] symbolic

executor. Selective symbolic execution successfully combines multi-path exploration with dynamic analysis of binary executables while lifting code to compiler-level IR. This makes S^{2}E a solid foundation for our binary recovery framework. On top of S^{2}E, we have implemented a plugin that logs the generated LLVM IR.

The logged IR is decoupled from the S^{2}E runtime state by our deinstrumentation engine, as exemplified in Figure 2. The engine implements the *disabled*, *error*, and *standard* modes of BinRec's fallback mechanism. When enabled, it directs all control flow edges with unknown targets in the logged IR to a trampoline. The trampoline copies lifted register values to physical registers before jumping to the corresponding code address in the original code of the binary.

5 EVALUATION

We evaluated BinRec by lifting binaries and running them on a system with 32GB RAM, and an Intel i7-6700 processor running at 3.4GHz with frequency scaling turned off. We used gcc 4.8.4 to compile programs to 32-bit x86 binaries.

We first evaluate the correctness and performance of a number of binaries from the SPEC-CPU2006 benchmarking suite, recovered by BinRec. Since KLEE does not support floating-point operations, S^{2}E implements these by emulation in integer operations. As this significantly degrades floating-point performance, we only evaluate the CINT subset of SPEC to obtain an accurate view of any instrumentation overhead.

Furthermore, we disable symbolic execution to assert that we evaluate correctness and performance of the recovered binary within the exact same code path followed by the original binary.

5.1 Correctness

We evaluated the correctness of the recovered binaries by running single-path binary recovery and comparing the behavior of the recovered binary with that of the input binary. Each program binary is lifted using its first reference input available in the SPEC-CPU2006 benchmark suite. If binary recovery is performed successfully and the same inputs are specified, the recovered binary must exercise exactly the same code paths as during recovery, not requiring the fallback mechanism. We use the **error** path of the fallback mechanism to test this behavior, and verify that the recovered and input binaries have the same output. We have compiled the 12 programs from the CINT set without optimizations and with full optimization (`-O0` and `-O3`), resulting in 24 input binaries. 9 of these do not work as expected for varying reasons:

- *omnetpp*, *gcc*, *xalancbmk* and *gobmk-O3* trigger incorrectly ordered execution events in S^{2}E, causing successor information to be recorded incorrectly. This leads to a fallback.
- *perlbench* triggers incorrect disassembly by S^{2}E due to a bug that causes an off-by-one error in the program counter.

The remaining 15 binaries all correctly follow the original code path, i.e., no fallback is detected. In order to measure the performance of the fallback mechanism, we have also recovered the remaining binaries with the fallback mechanism **disabled**, and again verified the results. *bzip2-O3* crashes when the fallback mechanism is disabled, due to a nontrivial over-optimization bug. This leaves a total of 29 output binaries for our performance evaluation.

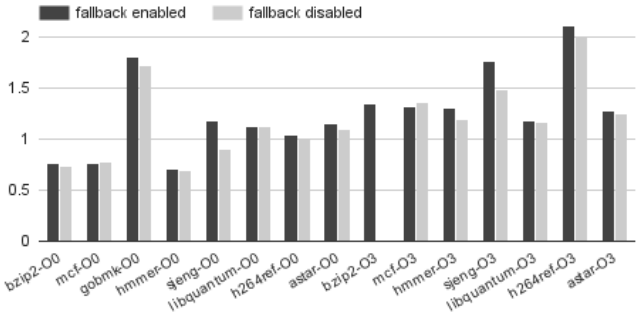

Figure 3: Normalized runtime of recovered SPEC2006 binaries (1 = input binary).

5.2 Performance

Figure 3 shows the runtime performance of recovered binaries compared to the respective input binaries. All recovered binaries have been optimized after lifting. For some unoptimized binaries, this leads to a significant speedup (*bzip2*, *mcf*, *hmmer* and *sjeng*, the latter only with fallback disabled). For most binaries, however, binary recovery incurs a slowdown. Manual inspection shows this is the case because LLVM optimizations cannot effectively track variable values on the stack, and thus keep them in memory rather than in registers after recompilation. The geometric mean of normalized runtimes for unoptimized (O0) binaries are 1.02 and 0.97 with the fallback mechanism enabled and disabled respectively. For optimized (O3) binaries, the geometric means are 1.44 and 1.39 respectively. We believe this to be an acceptable baseline instrumentation overhead given that (i) existing binary rewriting also introduce overhead but without allowing for IR-level transformations [34] and (ii) additional optimizations unique to our particular setting (e.g., profile-guided optimizations) are possible.

5.3 Attack Surface Reduction

Although progress is being made at quantifying the attack surface of the kernel [22], we are not aware of any universally accepted attack surface reduction metrics for user-space code. We therefore chose to measure the attack surface of the recovered programs by calculating the fraction of original program instructions that are lifted to IR code, and by comparing the number of ROP gadgets in the recovered code with the number of gadgets in the original code. Table 1 shows our findings. We evaluated only benchmarks that can be correctly recovered by BinRec; all input binaries are optimized (O3). Each program binary is lifted using all reference inputs available in the SPEC CPU 2006 benchmark suite. We then measured geometric means of results from different reference inputs. *astar*, for example, has two reference inputs *BigRakes2048* and *rivers*, thereby two recovered binaries were generated and measured for this benchmark. Using BinRec, only 28% of the original instructions are lifted to LLVM IR on average. We measured ROP gadget reduction in the recovered binaries using a ROP gadget finder tool called Ropper [2]. We found that the BinRec recovered binaries contain 48% fewer ROP gadgets than the original binaries.

Table 1: Attack surface reduction

Benchmark	% recovered instructions	ROP gadgets		
		# original	# recovered (gmean)	% reduction
astar	49.09%	1029	804	21.87%
bzip	55.91%	1070	581.67	45.64%
gobmk	19.70%	20564	4583.6	77.71%
h264ref	21.81%	9035	2315.67	74.37%
hmmer	8.17%	5488	1100	79.96%
libquantum	26.92%	1397	179	48.53%
mcf	57.20%	549	433	21.13%
sjeng	25.84%	2269	1013	55.35%
gmean	28.05%			47.56%

6 RELATED WORK

BinRec is not the first to address the problem of binary rewriting or code lifting. We now summarize related work.

Attack surface reduction. Previous work on attack surface reduction primarily targets the Linux kernel. Not all of these works targeted attack surface reduction specifically, but achieved it nonetheless as a side effect of code compaction. Lee et al., for example, constructed call graphs to identify dead code in the kernel, and then manually pruned this code at the source level [24]. Chanet et al. used a variety of static analyses, including constant propagation, to specialize the kernel for a given kernel command line [9]. Contrary to the aforementioned work by Lee et al., this solution operates at link time and is fully automated. He et al. analyze and compact the kernel at the source code level, and propose to leverage approximate decompilation and FA-analysis to incorporate the effects of hand-written assembly snippets into static analysis results [18]. More recently, Kurmus et al. used dynamic tracing to determine which kernel functionality is necessary to support a given system workload [22]. They then correlated the trace information to source lines, and subsequently calculated the minimal set of compile-time configuration options that must be enabled to generate a kernel that incorporates all the necessary functionality.

Low-level binary analysis and rewriting. Many projects target the problem of low-level binary analysis and rewriting. PEBIL [23], UQBT [13] and Uroboros [34] all statically rewrite binary programs either at the machine code level or using a custom low-level IR. Their main aim is to support the insertion of simple instrumentation where efficiency is more important than the ability to perform complex code transformations, such as altering the CFG. BISTRO [14] allows extracting and embedding functional components within binaries, and when combined with an analysis and concretization scheme such as one proposed by [20] can provide binary feature customization. While it targets advanced analysis, rather than code rewriting (as we do)), `angr` [31] bundles numerous static and dynamic analysis techniques, including symbolic execution.

Earlier work such as ATOM [17], PLTO [28], Diablo [27], and Vulcan [32] are powerful tools, but to our knowledge they do not work well with stripped binaries. Also, like dynamic instrumentation tools, they typically do not support a generic compiler-level IR. Dynamic instrumentation tools like PIN [25], Dyninst [6], DynamoRIO [1] and Valgrind [26] are dynamic binary analysis tools, providing runtime APIs to analyse and instrument code at runtime. They do not use a compiler-level IR, and since they lack the recompilation phases, they are weaker in applying advanced analysis or optimization passes.

Binary code lifting. LLBT [29, 30] statically retargets binaries to different ISAs after lifting them to LLVM IR. MC-Semantics [16], Dagger [5] and RevNIC [10] (which is based on S^2E) raise machine code for the purpose of high-level static binary translation on LLVM IR. SecondWrite [3] successfully recompiles real-world binaries after statically lifting them to LLVM IR, rejuvenating them by applying existing compiler optimizations. Such projects are limited by their static analysis and cannot handle complex program code such as highly optimized binaries, special idioms, etc. They are all based on static analysis. In addition, even the most advanced solution, SecondWrite, currently cannot cope with position-independent code. Finally, HQEMU [19] extends the QEMU code generation back-end to lift code to LLVM IR similarly to S^2E (and therefore BinRec), for the purpose of optimization. It does not decouple lifted code from the QEMU runtime to produce a standalone executable binary, like BinRec.

7 CONCLUSION

We presented BinRec, a new solution for attack surface reduction based on binary analysis and rewriting. BinRec recovers only an essential part of the program by capturing execution traces and discovering more execution paths reachable through symbolic execution. Our experiment shows that BinRec successfully removes over 70% of the original instructions while preserving the program's functionality, and that the BinRec recovered binaries contain nearly 50% less ROP gadgets. To implement a conservative recompilation strategy, BinRec relies on fallback code to handle residual binary code that did not execute during profiling. Since the program rarely reaches the fallback code, we can apply slower yet stronger hardening solutions to the fallback code, e.g., precise control-flow integrity instrumentation. Applying such hardening techniques to fallback code will allow us to further reduce the attack surface.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", in part by the United States Office of Naval Research (ONR) under contracts N00014-17-1-2782 and N00014-17-S-B010 "BinRec", in part by the European Commission (Horizon 2020 - DS-07-2017) under Grant #786669 "ReAct", in part by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, and in part by the National Science Foundation under awards CNS-1619211 and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of any of the sponsors or any of their affiliates. The authors also gratefully acknowledge a gift from Oracle Corporation.

REFERENCES

[1] DynamoRIO. http://dynamorio.org.
[2] Ropper. https://scoding.de/ropper/.
[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Eurosys*, 2013.
[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC*, 2005.
[5] Ahmed Bougacha, Geoffroy Aubey, Pierre Collet, Thomas Coudray, Jonathan Salwan, and Amaury de la Vieuville. Dagger decompiling to ir. 2013.
[6] Bryan Buck and Jeffrey K Hollingsworth. An api for runtime code patching. *IJHPCA*, 2000.
[7] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[8] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, 2011.
[9] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide compaction and specialization of the linux kernel. *ACM SIGPLAN Notices*, 2005.
[10] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *EuroSys*, 2010.
[11] Vitaly Chipounov and George Candea. Enabling sophisticated analyses of x86 binaries with revgen. In *DSN-W*, 2011.
[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. *S2E: a platform for in-vivo multi-path analysis of software systems.* 2012.
[13] Cristina Cifuentes and Mike Van Emmerik. Uqbt: Adaptable binary translation at low cost. *Computer*, 2000.
[14] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: Binary component extraction and embedding for software security applications. In *Computer Security – ESORICS 2013*, 2013.
[15] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *CC*, 2017.
[16] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon*, 2014.
[17] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *USENIX TCON*, 1995.
[18] Haifeng He, John Trimble, Somu Perianayagam, Saumya Debray, and Gregory Andrews. Code compaction of an operating system kernel. In *CGO*, 2007.
[19] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *CGO*, 2012.
[20] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *ICSE*, 2014.
[21] James C King. Symbolic execution and program testing. *CACM*, 1976.
[22] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *NDSS*, 2013.
[23] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS*, 2010.
[24] Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. An application-oriented linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 2004.
[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *SIGPLAN*, 2005.
[26] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *SIGPLAN*, 2007.
[27] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *ISSPIT*, 2005.
[28] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the intel ia-32 architecture. In *WBT*, 2001.
[29] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. Llbt: an llvm-based static binary translator. In *CASES*, 2012.
[30] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. A retargetable static binary translator for the arm architecture. *TACO*, 2014.
[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P*, 2016.
[32] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.
[33] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 866–879. IEEE, 2015.
[34] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *USENIX SEC*, 2015.