Vertx: Automated Validation of Binary Transformations*

Denis Gopan
GrammaTech, Inc.
gopan@grammatech.com

Peter Ohmann†
Xavier University
ohmannp@xavier.edu

David Melski
GrammaTech, Inc
melski@grammatech.com

ABSTRACT

The paper describes Vertx, a tool for validating software binary transformations. Vertx enables software developers and system administrators to automatically check the correctness of software binary transformations such as security hardening and optimization. Transformation validation increases user trust in binary-transformation technology and allows binary transformations to be safely used in critical applications that have narrow margins of error.

1 INTRODUCTION

Technology for transforming software binaries provides many unique advantages to software developers, security engineers, and system administrators. It allows them to modify legacy software or commercial of-the-shelf (COTS) components—for which source code is often unavailable—to incorporate general security protections, specific vulnerability fixes, and runtime monitoring of application-specific policies. Additionally, software binaries can be optimized and specialized with respect to their operational environment to reduce their sizes and/or improve their efficiency. Even when source code is available, transforming software at the machine-code level may be advantageous because it avoids the WYSINWIX ("What You See [in the source code] Is Not What You eXecute") phenomenon [3–5].

Automatically modifying software binaries is a highly complicated undertaking. Binary transformations must leverage many sophisticated analyses and tools to: (1) accurately recover the high-level *Intermediate Representation* (IR) from the binary; (2) soundly modify the IR to achieve the transformation goals; and (3) correctly reassemble the IR back into an operational binary. A slight error or imprecision in any step may cause the modified software to malfunction. Often, software patches that fix one set of errors simultaneously introduce new errors and vulnerabilities [13, 14, 16, 20, 21]. As a result, the users are faced with a dilemma: either use existing bloated and insecure software, or risk further breaking the software in an attempt to fix it with an untrusted binary transformation. In practice, reliability-conscious users often elect to avoid using binary transformation tools.

In this paper, we present Vertx, a tool that aims to enable the application of binary transformations in a safe and verifiable manner. We took inspiration for Vertx from the translation-validation techniques that were designed to validate the correctness of optimizing compilers [12, 15, 17]. Rather than attempting to prove the overall correctness of compiler implementation, translation validation techniques view the compiler as untrusted, and instead focus on validating each individual compilation. They compare the original program to the optimized one to see if there are any semantic differences. Such differences indicate that the compiler is buggy. To achieve scalability, translation validation leverages the fact that the two programs are similar structurally and rely on the information emitted by the compiler to understand how the program was transformed.

Vertx operates in a similar fashion. However, there are several important differences to the translation-validation setting that Vertx must reckon with:

- Vertx targets general software transformations, such as security hardening, which do not preserve the exact semantics of the original program. Instead, such transformations augment the semantics of the original program to exclude *unsafe* behaviors.
- Vertx operates directly on software binaries and must accurately model low-level details such as code and data layout. In contrast, existing translation validation approaches operate on high-level intermediate representation maintained by a compiler.

In essence, Vertx constitutes the trusted computing base (TCB) of a binary-transformation framework. Vertx is small, makes a minimal number of necessary assumptions about the software and its execution environment, and is built around general, well-exercised components. Thus, assessing the correctness of Vertx is much easier than establishing the correctness of the entire transformation framework.

We built a prototype of Vertx and used it to successfully validate several useful security transformations. Currently, Vertx only supports transformations that (i) do not impose global changes on program semantics, and (ii) preserve the layout and placement of data. While limited, this class of transformations includes, for instance, control flow integrity

*This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. W31P4Q-14-C-0083 and Department of Homeland Security (DHS) under Contract No. HSHQDC-16-C-00099. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA and DHS.

†The work was performed during the author's internship at GrammaTech, Inc.

FEAST'17, November 3rd, 2017, Dallas, TX, USA.

ACM ISBN ISBN 978-1-4503-5395-3/17/11...$15.00
https://doi.org/10.1145/3141235.3141237

enforcement [1, 2], which has been shown to be useful in practice.

Contributions. Our work makes the following contributions:

- We reduce the TCB of a general framework for transforming software binaries.
- We adapt translation-validation techniques to effectively validate binary transformations that modify the semantics of the software.
- We adapt translation-validation techniques to directly operate on software binaries.

Paper Organization. The paper is organized as follows. §2 gives an intuitive overview of our approach. §3 describes transformation validation more formally. §4 presents experimental evaluation of VERTX. §5 discusses related work.

2 OVERVIEW

Fig. 1 shows how VERTX fits into the overall software-binary transformation ecosystem. VERTX provides a trusted "checker" to validate any changes made by *any* untrusted binary transformation tool. It leverages formal verification-style techniques to ensure that:

- The transformation achieves its stated goal (e.g., eliminates *unwanted* program behaviors).
- The transformation does not break software functionality (i.e., preserves its *desired* behaviors).

VERTX is a stand-alone tool with a well-defined interface. It takes several inputs:

- Two versions of the binary: the original and the transformed;
- A *trusted* transformation specification capturing the intent of the transformation;
- *Untrusted* transformation hints to enable scalable validation;

and produces a *yes* or *no* answer. The "yes" answer indicates that the transformation is correct with respect to the specification. The "no" answer indicates that either (i) the transformation does not adhere to the specification (that is, the transformation is buggy and needs to be fixed), or (ii) the transformation is correct, but the hints are insufficiently strong/precise to enable the validation. It is possible for some of the validation tasks to timeout, also resulting in the "no" answer. To help the users determine the causes for the "no" answer, VERTX generates a detailed validation report.

We expect that, in most cases, the transformation specification and hints will be generated automatically by the binary-transformation tool rather than being authored manually by the user. We believe that this places only a small burden on most transformation tools. It is likely that the tools already compute the information required for validation as part of the transformation.

Currently, the hints that VERTX requires from a transformation tool consist of a map that captures the correspondence between the code fragments in the original program and their transformed counterparts. The map is from the effective code addresses in the original program to addresses in the transformed programs. The next section describes the information VERTX requires in greater detail. In the future, as we add support for additional classes of general binary transformations, we expect that VERTX will require more sophisticated set of hints.

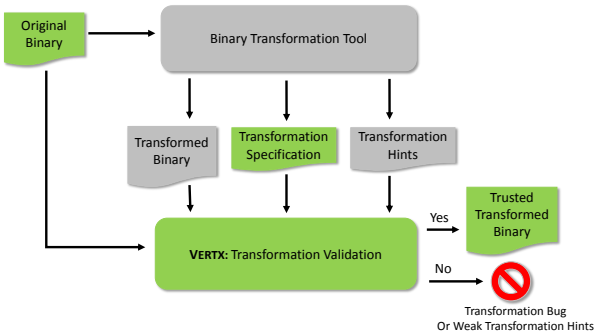

Figure 1: The software-binary transformation ecosystem. Untrusted components and artifacts are shown in gray; trusted ones are shown in green.

3 TRANSFORMATION VALIDATION

The purpose of transformation validation is to (i) make sure that the transformation achieves its aims, and (ii) that the correct semantics of the program is not modified. Our approach is inspired by translation-validation techniques [15, 17]. We partition the programs (both original and transformed) into sets of *translation blocks*, and for each pair of corresponding blocks (a block from the original program and its counterpart from the transformed program), we check that:

- The transformed block does not cause the violation of the enforced property
- The behaviors of two blocks are equivalent under the assumption that the enforced property holds.

A translation block is *not* a straight-line single-entry, single-exit sequence of instructions (i.e., not a basic block). Rather, translation block boundaries are defined by a set of cut-points on an inter-procedural control flow graph. Thus, translation blocks are single-entry and multi-exit instruction sequences that may contain control flow paths that split and rejoin. However, blocks are not allowed to have cycles (VERTX relies on symbolic execution to capture block's semantics and thus requires blocks to have a bounded number of behaviors). Also, blocks must stop at system calls (system calls contribute to observable program behaviors; thus, VERTX must ensure that they are invoked from equivalent program states).

3.1 Validation Algorithm

In this section, we describe key aspects of our validation mechanisms. Below, let E_a denote the set of effective code addresses, and let ϕ denote a set of logical formulas over a program state (i.e., in our machine-code analysis setting, ϕ can reference registers, flags, and memory addresses).

High-level Overview. VERTX takes the following inputs:

- The original and transformed programs, in binary form.
- A correspondence map, $M_{corr} : E_a \rightarrow E_a$, that maps starting point of each translation block in the original program to the corresponding block in the transformed program.
- A property $\Phi : E_a \rightarrow \phi$ that the transformation aims to enforce. Φ is interpreted as follows: for each $e \in E_a$ in the domain of Φ, $\Phi(e)$ must hold at program point e.
- An error-handler address, $e_{hndlr} \in E_a$: an address in the transformed program to which the control is transferred when the property is violated.
- A *scratch space*, S_{tx}: a memory area that the introduced instrumentation uses for its purposes.

For each effective-address binding in M_{corr}, the tool:

(1) Extracts the corresponding translation blocks from the original and transformed programs and logically captures their semantics.
(2) Checks that, for the executions of the blocks that comply with Φ, the semantics is equivalent.
(3) Checks that any execution of the transformed block that violates Φ is redirected to the designated error handler, e_{hndlr}.

The transformation is declared to be valid only if the process succeeds for all bindings in M_{corr}.

Translation Block Extraction. A translation block is identified by its starting effective address. To extract and logically capture the behavior of the block, we rely on symbolic execution. The symbolic state is propagated down each path reachable from the starting point. The propagation stops when one of the following conditions is met:

- The propagation reaches an effective address $a \in E_a$ that is mapped by the correspondence map. That is, either $a \in domain(M_{corr})$ for the original program, or $a \in range(M_{corr})$ for the transformed program.
- The propagation reaches a call to an external function (i.e., a function that resides in a shared library).
- The propagation reaches a system call.

The propagation stops at the external (library) function calls and system calls because those calls are used to interact with the environment and generate observable program events. The translation validation must ensure that those calls are performed in the states that are similar in both programs and the semantic state equivalence is only established at block boundaries.

The symbolic states that are propagated along each explored path are tuples, $\langle C, S\rangle$, where C is the path constraint and S is the transformation map that keeps symbolic values for processor registers, flags, and memory. The resulting logical characterization for the code block is a set of such tuples with one tuple for each execution path in the block. To keep the block exploration bounded, the blocks are not allowed to contain loops. If a loop is encountered, the validation of the block and, consequently, of the entire program fails.

If the propagation encounters an indirect branch, a decision procedure is used to enumerate the possible control flow targets. The enumeration is performed up to a user-specified limit. If the limit is exceeded, a warning is issued to the user. In that case, it is up to the user to decide whether to trust the validation results. A failed target enumeration may indicate that the control can be transferred to a code fragment that is not covered by the correspondence map, and thus is not exposed to the validation tool. Those fragments can harbor semantic differences that went unnoticed. However, if the user is confident that the correspondence map is complete (achieves full code coverage), she may still trust the validation results.

Logical Encoding of Property Φ. VERTX only compares the blocks for semantics equivalence for the executions that satisfy the property Φ. To logically characterize those executions, VERTX translates global property Φ to a set of translation-block specific properties Φ_B, one for each translation block B in a program. Essentially, instead of tracking the effective addresses of the program points where the property is specified, Φ_B captures the path constraints that reach those program points during the execution of B. The translation is done as part of symbolic execution: whenever a program point $e \in domain(\Phi)$ is reached with symbolic state $\langle c, S\rangle$, a mapping $[c \rightarrow \Phi(e)]$ is added to Φ_B.

Fig. 2 (d) shows the pseudo-code for encoding Φ_B. The `dp_ctx` is a decision procedure context, and SS is the set of symbolic states computed for the block B. The notation $\phi\,[S]$ in `encode_formula` denotes evaluation of ϕ over a symbolic value map S (that is, the leaf terms in ϕ that correspond to registers, flags, and memory are replaced with the corresponding symbolic values from S).

Comparing Block Semantics. VERTX checks the equality of each register, flag, and memory separately. For the blocks to be equivalent, each individual comparison must succeed. Fig. 2 (f) shows the pseudo code for checking the equivalence for an arbitrary processor register or flag, x. The check starts by asserting that the property holds (to ensure that only the *correct* executions are taken into consideration). Then the symbolic values for the register (flag) in the original and transformed programs are encoded with the use of `encode_reg_flag` primitive (Fig. 2 (a)). Finally, a decision-procedure query is issued that checks whether the two symbolic values can differ. If the query is unsatisfiable, then the two values are guaranteed to be the same.

The memory equivalence check shown in Fig. 2 (g) is similar, except that instead of comparing the memory in its entirety (a memory is represented as a logical array), we non-deterministically select a memory address and check the equivalence of the values stored at that address. In essence, we ask a decision procedure to find an address where the memory contents may differ. If no such address exists, the memory contents ought to be equivalent. The introduced instrumentation may need to compute and store intermediate data (e.g., spill register values). VERTX accounts for this by allowing the transformation to specify a *scratch space*, M_{tx},

```
Expr encode_reg_flag(dp_ctx, x, SS = {⟨c, S⟩}) {
    Expr v_x = dp_ctx.fresh_var()
    foreach ⟨c, S⟩ ∈ SS {
       dp_ctx.assert(c ⇒ (v_x = S(X)))
    }
    return v_x
}
```

(a)

```
Expr encode_mem_access(dp_ctx, a, SS = {⟨c, S⟩}) {
    Expr v_a = dp_ctx.fresh_var()
    foreach ⟨c, S⟩ ∈ SS {
       dp_ctx.assert(c ⇒ (v_a = read(S(Mem), a)))
    }
    return v_a
}
```

(b)

```
Formula encode_formula(φ, SS = {⟨c, S⟩}) {
    return ⋀_{⟨c,S⟩∈SS}(c ⇒ φ[S]))
}
```

(c)

```
bool check_property(Φ_B, SS_tx) {
    DP dp_ctx = fresh_contex()
    Expr v_ip = encode_reg_flag(dp_ctx, R_ip, SS_tx)
    Formula f =
          ⋁_{c↦φ∈Φ_B}(c ∧ ¬ψ ∧ (v_ip ≠ e_hndlr))
                where ψ = encode_formula(φ, SS_tx)
    return (dp_ctx.check(f) == unsat)
}
```

(e)

```
void encode_property(dp_ctx, Φ_B, SS = {⟨c, S⟩}) {
    foreach c ↦ φ ∈ Φ_B {
       Formula f_φ = encode_formula(φ, SS)
       dp_ctx.assert(c ⇒ f_φ)
    }
}
```

(d)

```
bool compare_memory(SS_orig, SS_tx, Φ_B, M_tx) {
    DP dp_ctx = fresh_context()
    encode_property(dp_ctx, Φ_B, SS_orig)
    Expr a = dp_ctx.fresh_var()
    dp_ctx.assert(a ∉ M_tx)
    Expr v_orig = encode_mem_access(dp_ctx, a, SS_orig)
    Expr v_tx = encode_mem_access(dp_ctx, a, SS_tx)
    return (dp_ctx.check(v_orig ≠ v_tx) == unsat)
}
```

(g)

```
bool compare_reg_flag(x, SS_orig, SS_tx, Φ_B) {
    DP dp_ctx = fresh_context()
    encode_property(dp_ctx, Φ_B, SS_orig)
    Expr x_orig = encode_reg_flag(dp_ctx, x, SS_orig)
    Expr x_tx = encode_reg_flag(dp_ctx, x, SS_tx)
    return (dp_ctx.check(x_orig ≠ x_tx) == unsat)
}
```

(f)

Figure 2: Pseudo-code for validating the transformation of a single translation block. Notation: SS_{orig} and SS_{tx} denote sets of symbolic states obtained for the original and transformed programs, respectively; R_{ip} denotes the instruction-pointer register for the target architecture (e.g., `%eip` on x86); $\phi[S]$ denotes an evaluation of formula ϕ over the map of symbolic values S.

a set of data locations that are only used by the transformed program. The pseudo-code in Fig. 2 (g) explicitly excludes addresses in M_{tx} from comparison.

Checking Φ for the Transformed Block. The last check that VERTX performs is ensuring that the transformed block does not violate the property Φ. Fig. 2 (e) shows the pseudo-code for the check. The constructed formula contains a disjunct for each potential property violation in the block. Each disjunct is a conjunction of:

- A precondition for reaching a program point where the property is checked,
- A condition for property violation (the negation of property encoding),
- A check that control did not reach the designated error handler.

For a given disjunct to be true, the following conditions must be met: (i) the precondition for reaching the sensitive program point must be satisfied, (ii) the property at that program point must be violated, and (iii) the control must not be redirected to the error handler. If the overall formula is unsatisfiable, then every property violation in the block results in the control to be routed to the error handler.

3.2 Validation Challenges

In this section, we describe challenges that we encountered when building VERTX.

Code Coverage. To establish transformation correctness, VERTX must ensure that *all* of the code in both programs is properly compared. To ensure that, VERTX performs several additional checks:

- The entry points of both programs must be mapped by M_{corr}.
- The return points of all external function calls and system calls must be mapped by M_{corr}.
- The targets of all indirect and computed control flow transfers must be mapped by M_{corr}

The first check is done before the individual block comparison begins. The second and third checks are done during during block extraction with the help of a decision procedure.

As we mentioned in §3.1, sometimes VERTX will not be able to fully enumerate the targets of an indirect control

flow transfer, and thus ensure that a complete code coverage is attained. To make a transformation fully verifiable, it is advantageous to include an enforcement of control flow integrity (CFI) [1, 2] as a part of it. CFI ensures that all control transfers in a program target *legal* destination addresses. Typically, a CFI policy white-lists a set of legal destination addresses for each indirect control transfer in a program. When CFI policy is available, VERTX ensures that all legal target addresses in the policy are mapped by M_{corr}.

There is one scenario for which code coverage cannot be checked mechanically: the call-back addresses that are passed to external functions and system calls must also be contained in M_{corr}. However, it is not trivial to identify such callback addresses without knowing the semantics of the called functions. Currently, we handle this issue by providing models for some standard functions of interest. However, having a more general approach is desirable.

Changes in Code Layout. Most useful program transformations will affect the layout of the code in software. The change in code addresses causes the following problem: the registers and memory locations that hold the addresses cannot be compared directly anymore. For example, consider an instruction `call <foo>` that is located at different effective addresses in the original and transformed programs (let's assume that the address of `foo` is the same for both programs). The original code and the transformed code are syntactically and semantically equivalent, yet the program states will differ because the return address pushed onto the stack will be different in the two programs.

We address this issue by modeling the code addresses symbolically. Scalar code addresses that are mapped by M_{corr} are replaced in the logical encoding with unconstrained symbolic constants. Let $\alpha \mapsto \beta \in M_{corr}$. Our approach will substitute all appearances of α in the symbolic encoding of the original program with unconstrained symbol sym_α. Similarly, occurrences of β will be replaced with sym_β in the encoding of the transformed program. Before performing decision procedure queries, additional constraints that restrict possible values of the introduced symbols are added to the logical context. Depending on the context of the query, either constraint $sym_\alpha = sym_\beta$ or $sym_\alpha = \alpha \wedge sym_\beta = \beta$ are added. The former allows VERTX to properly compare code addresses that changed as the result of the transformation, while the latter covers the cases in which the actual values of code address are important (e.g., the sieve transform described in §4, which explicitly translates original code addresses to their transformed counterparts).

Currently, VERTX only handles transformations that preserve data placement and layout; thus, we did not yet address the corresponding set of issues for the data layout. Supporting transformations that perturb data layout will be the focus of the future work.

Error-Handler Integrity. VERTX may successfully prove that any violation of a targeted security property is properly routed to a designated error handler. However, VERTX does not check that the error handler, introduced by an untrusted rewriting tool, does not itself contain vulnerabilities or malicious code. The simplest approach to rectify this is to force the transformation tool to use a general, program independent error handler that can be compared verbatim to its original version. Such an error handler can be analyzed and verified independently to prove the absence of security vulnerabilities.

Sanity Constraints. Early in our experiments, we ran into a need to model certain *sanity constraints* about the global program-execution environment. Specifically, we needed to separate stack and data spaces. Without such separation, a decision procedure is able to pick addresses that overlap the stack with the data segment. As the result, any write to a global variable could be interpreted as the write into the stack frame (e.g., a stack-smashing attempt) causing the validation to fail. We addressed this issue by adding constraints on the values of the stack pointer and frame pointer that clearly separate the stack from the data (we picked a reasonable low bound for the stack addresses and added the constraints that ensured that the initial values of the stack and frame pointers for each code block are above that bound). However, this approach is overly simplistic. To provide true validity guarantees, VERTX must also ensure that such constraints can never be violated.

4 EXPERIMENTAL RESULTS

We built a prototype of VERTX and evaluated it on a number of real-world programs. We used the prototype to validate the *sieve transformation*, a transformation that modifies software to dynamically translate control-flow targets (code addresses) from the original binary to their counterparts in the transformed binary. Below, we describe our experimental setup and the results we obtained in detail.

Sieve Transformation. We used the VERTX prototype to validate the sieve transformation. Sieve is a mechanism that is used to effectively translate code addresses in the original software to their counterparts in the transformed software [11, 18]. Each indirect control-flow transfer instruction in the original software is prefixed with code that performs a look up in the table of allowable targets. If the target is in the table, the corresponding code address in the transformed executable is used as the target for the control transfer; otherwise, an exception is raised. The sieve transformation has two important practical uses. First, it enforces control-flow integrity [2], thus, preventing control hijacking attacks, such as stack smashing and return-oriented programming [7]. Second, the underlying mechanism simplifies the translation of the code-address computations, making the transforms that leverage it more robust.

We used a control-flow integrity (CFI) policy as the target property for the transformation. Essentially, a CFI policy specifies a set of legal targets for each indirect control-flow transfer in a program. We used our existing infrastructure for machine-code analysis to derive suitable CFI policies for our benchmarks.

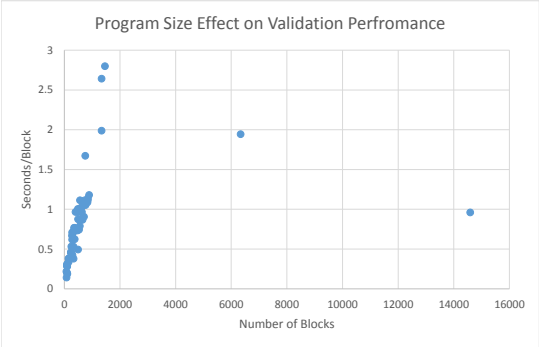

Figure 3: The effect of program size on validation efficiency.

Benchmarks. To evaluate VERTX, we selected a subset of challenge binaries used by DARPA in Cyber Grand Challenge (CGC) competition [19]. The CGC challenge-binary suite contains 183 programs of various size. The programs are 32-bit x86 statically-linked binaries that rely on CGC runtime support (CGC provides its own execution environment complete with a simplified operating system and a version of standard C library). Currently, the VERTX prototype does not support floating point and XMM instructions. Thus, we selected a subset of binaries (59 out of 183) that do not use those instructions. The selected binaries range in size from about 80K to 1.2M.

Setup. We ran the experiments in a virtual environment. The host machine was equipped with 6-core Intel Xeon CPU ES-2620 and 16G of RAM and run Windows 7. The guest environment was configured to use 2 processor cores and 8G of memory, and ran Ubuntu 14.04.5 LTS.

Results. VERTX was able to successfully validate the application of sieve transformation to the set of benchmarks we selected. In fact, VERTX helped us catch an error in the transformation implementation. The sieve transformation introduces a couple of global memory locations for storing intermediate computation results. The sizes of these locations were not updated when porting the sieve transformation from 32-bit to 64-bit. As the result, the values stored at those locations overlapped causing the software to crash. VERTX successfully detected and reported this issue.

We found the performance of the VERTX prototype to be adequate in practice. As we discussed in §3, the validation of each translation block is performed independently. Thus, the overall validation time is a linear factor of the time it takes (on average) to validate a single block transformation. Fig. 3 shows the relationship between software size (given in the number of translation blocks) and the average block validation time. On average, it takes just a few seconds per block, though block validation time increases with the increase in program size. We believe that this increase is specific to the sieve transformation: the larger the program, the more targets each sieve contains. Since the entire sieve code falls within a translation block, larger sieves translate into longer block validation times. The two largest benchmarks appear to be "outliers" with unexpectedly low block validation times Unfortunately, we did not have a chance to track the exact reason for this deviation.

Overall, it took the prototype a few minutes to validate smaller-size CGC challenge binaries (up to 200K). However, as the software size increased so did the validation times, rising up to several hours for validating several of the largest CGC challenge binaries we experimented with (500K to 1M). Even if block validation takes just a second, processing 14K blocks adds up to almost 4 hours.

To see how VERTX handles large real-world software, we experimented with applying it to Apache web server (22K translation blocks). Our observations agreed with what we saw in the CGC challenge-binaries experiments—the average block validation time was further increased due to the increase in the program size. In the future work, we plan to focus on cutting down block validation times to make the tool more scalable.

5 RELATED WORK

The problem of transformation validation was first addressed in the context of compiler correctness. There are two general approaches to this problem: (i) verifying the correctness of the transformation (compiler) implementation, and (ii) checking the validity of each individual transformation (compilation). An example of the former approach is CompCert project [12]. An example of the latter approach is translation validation [10, 15, 17].

VERTX is modeled after the latter approach because we believe it is more practical: (i) it offers better scalability by reasoning at the level of individual translation blocks, (ii) it does not impose restrictions on the development of transformation tools (such as, using Coq [6]), and (iii) it allows transformation tools to contain latent errors as long as those errors do not compromise the correctness of specific transformations. In this work, we extended the translation-validation approach to apply to transformations that augment software semantics, and designed mechanisms for applying it to software binaries directly (as opposed to higher-level program representations).

Recently, the area of Incremental Verification (IV) has gained popularity [8, 9]. IV aims to reuse efforts between verification runs. It assumes that the original program, P, has been verified with respect to a certain property ϕ, and attempts to efficiently prove that ϕ also holds for program Q, which is derived from P. To accomplish this, IV techniques build an abstract simulation relation between P and Q that can be used to lift the proof of ϕ in P to also work for Q. This approach does not directly match the problem we are addressing, but it bears a strong similarity to the VERTX concept of operation. It remains to be seen whether it is possible to adapt some aspects of IV techniques to work effectively in our context.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, 2004.

[4] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), Aug. 2010.

[5] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Zurich, Switzerland, 2005/10/10/October 10 2005. Springer. Zurich, Switzerland.

[6] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*, 1997.

[7] S. Checkoway, J. Halderman, A. J. Feldman, E. W. Felten, B. Kantor, and H. Schacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *EVT*, 2009/// 2009.

[8] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Incremental verification of compiler optimizations. In *NASA Formal Methods*, pages 300–306, 2014.

[9] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Property directed equivalence via abstract simulation. In *Int. Conf. on Computer Aided Verification (CAV)*, pages 433–453, 2016.

[10] B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, 2005.

[11] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73. IEEE Computer Society, 2007.

[12] X. Leroy. Formal verification of a realistic compiler. *Comm. of the ACM*, 52(7):107–115, 2009.

[13] R. McMillan. Intel Reissues Buggy Patch. http://www.pcworld.com/article/126918/article.html, February 2006.

[14] R. McMillan. After buggy patch, criminals exploit Windows flaw. http://www.infoworld.com/article/2627362/hacking/after-buggy-patch--criminals-exploit-windows-flaw.html, June 2010.

[15] G. C. Necula. Translation validation for an optimizing compiler. In *Conf. on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices, pages 83–94. ACM, June 2000. Vancouver, British Columbia, Canada.

[16] R. Pegoraro. Apple Updates Leopard—Again. http://voices.washingtonpost.com/fasterforward/2008/02/apple_updates_leopardagain.html, February 2008.

[17] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 151–166, 1998/// 1998.

[18] S. Sridhar, J. S. Shapiro, and P. P. Bungale. Hdtrans: A low-overhead dynamic translator. In *Workshop on Binary Instrumentation and Applications (WBIA)*, pages 135–140, Seattle, Washington, 2005/09/18/ 2005. IEEE Computer Society. St. Louis, MO.

[19] M. Walker. Cyber grant challenge (CGC). http://www.darpa.mil/program/cyber-grand-challenge.

[20] L. Whitney. Mcafee to compensate home users for bad update. https://www.cnet.com/news/mcafee-to-compensate-home-users-for-bad-update/, April 2010.

[21] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *Foundations of Software Engineering*, pages 26–36, 2011.