TASE: Reducing Latency of Symbolic Execution with Transactional Memory

Adam Humphries
University of North Carolina
humphries@cs.unc.edu

Kartik Cating-Subramanian
University of Colorado
kartik@simplescientist.net

Michael K. Reiter
Duke University
michael.reiter@duke.edu

***Abstract*—We present the design and implementation of a tool called TASE that uses transactional memory to reduce the latency of symbolic-execution applications with small amounts of symbolic state. Execution paths are executed natively while operating on concrete values, and only when execution encounters symbolic values (or modeled functions) is native execution suspended and interpretation begun. Execution then returns to its native mode when symbolic values are no longer encountered. The key innovations in the design of TASE are a technique for amortizing the cost of checking whether values are symbolic over few instructions, and the use of hardware-supported transactional memory (TSX) to implement native execution that rolls back with no effect when use of a symbolic value is detected (perhaps belatedly). We show that TASE has the potential to dramatically improve some latency-sensitive applications of symbolic execution, such as methods to verify the behavior of a client in a client-server application.**

I. Introduction

Since its introduction [7], [29], symbolic execution has found myriad applications for security analysis and defense (e.g., [31], [9], [11], [52], [20], [51], [35], [46], [41], [57]), software testing (e.g., [50], [25], [43], [49], [2], [40], [26], [12]), and debugging (e.g., [56], [54]). Whereas regular, "concrete" execution of a program maintains a specific value for each variable, symbolic execution allows some "symbolic" variables to be undetermined but possibly constrained (e.g., to be in some range). Upon reaching a branch condition involving a symbolic variable, each branch is executed under the constraint on the symbolic variable implied by having taken that branch. Any execution path thus explored yields a set of constraints on the symbolic variables implied by having taken that path. In an example use case, these constraints could be provided to an SMT solver [36] to compute a concrete assignment to the symbolic inputs that would cause that path to be executed.

When applied to testing, the speed of symbolic execution is typically a secondary concern. However, several security applications place symbolic execution on the critical path of defensive response in time-critical circumstances. For example, some works (e.g., [9], [20]) leverage symbolic execution to generate vulnerability signatures upon detecting an exploit attempt, and so the speed of symbolic execution is a limiting factor in the speed with which vulnerability signatures can be created and deployed to other sites. Other examples are intrusion-detection systems in which a server-side *verifier* symbolically executes a client program to find an execution path that is consistent with messages received from the client, without knowing all inputs driving the client (e.g., [19], [16]). If each message could be verified before delivering it to the server, then the server would be protected from exploit traffic that a legitimate client would not send (e.g., Heartbleed packets to an OpenSSL server [16]). However, such tools are not yet fast enough to perform this checking on the critical path of delivering messages to the server, reducing them to detecting exploits *alongside* server processing.

Conventional wisdom holds that SMT solving and state explosion are the primary latency bottlenecks in symbolic execution. However, the speed of straightline, concrete execution has been found to be the primary culprit in some contexts (e.g., [19], [55]). Most symbolic execution tools incur a substantial performance penalty to straightline execution because they interpret the program under analysis, even when it is performing operations on concrete data. For example, Yun et al. [55] report straightline-execution overheads of 3000× and 321,000× native execution speed for KLEE [10] and angr [46], due to interpretation. The need to interpret the program in these tools arises from the need to track symbolic variables, to accumulate constraints on those variables along each execution path, and to explore multiple execution paths. Even attempts to optimize symbolic execution when processing only instructions with concrete arguments must typically incur overheads due to lightweight interpretation; e.g., S^2E [18] encounters an overhead of roughly 6× vanilla QEMU [4] execution speed on purely concrete data due its use of memory sharing between QEMU and KLEE. In our microbenchmark evaluations (see Sec. VI-A), these overheads in S^2E resulted in concrete execution costs of up to ≈ 72× native execution.

In this paper, we provide a solution that supports fast native execution of instructions with concrete values—while still using interpretation to do the "symbolic parts" of symbolic execution—on modern x86 platforms. Our design and associated tool, called TASE[1], accomplish this through two key innovations. Though TASE instruments the executable to test whether variables are concrete or symbolic (like EXE [11]), our first innovation amortizes the costs of these tests by batching many into a few instructions. To maximize the benefits of this amortization, TASE defers these checks to ensure

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24327
www.ndss-symposium.org

[1]TASE stands for "Transactional Acceleration for Symbolic Execution".

that only variables actually used are checked; this deferment, together with the amortization, means that instructions may be concretely executed on symbolic variables. Therefore, a critical second innovation in TASE is a way of rolling back such erroneous computations so they have no effect. TASE uses hardware transactions as supported by Intel TSX extensions for this purpose, though as we will see, accomplishing with low overhead is nontrivial.

This paper outlines the design of TASE, and evaluates its potential to accelerate symbolic execution of applications with small amounts of symbolic state. We first show where TASE improves over modern alternatives such as KLEE, S^2E, and QSYM [55], through a microbenchmark comparison. This comparison shows that while TASE can perform poorly relative to some of these alternatives for applications with large amounts of symbolic data, it can perform much better than them when the amount of symbolic data is small.

Second, we show how TASE qualitatively improves the deployment options for a specific defensive technique, namely behavioral verification of a client program [19], [16] as introduced above. Though Chi et al. were able to show the verification of OpenSSL client messages in TLS 1.2 sessions induced by a Gmail workload at a speed that coarsely keeps pace with these sessions [16], their verification was not fast enough to perform on the critical path of message delivery. We show that replacing the symbolic execution component of their tool with TASE substantially improves the prospects for performing verification as a condition of message delivery. More specifically, we show that TASE's optimizations reduce the average, median, and maximum *lag* suffered by any client-to-server message by over 90%, 94%, and 79%, where lag is defined as the delay between arrival of a message to the verifier and its delivery to the server after verification completes. In doing so, TASE brings these lags into ranges that are practical for performing inline verification of TLS sessions driven by applications, like Gmail, that are paced by human activity.

Third, we demonstrate the flexibility of the techniques in TASE by using it to prototype memory protections (stack canaries and buffer under- or over-run detection, while either reading or writing) for program execution. This demonstration is of interest primarily due to its minimality, requiring changes to TASE and associated components of fewer than 140 source lines of code. The resulting protections add less than 15% to our microbenchmarks' execution times with TASE, as well.

To summarize, our contributions are as follows:

- We introduce a method to limit tests for determining whether variables are symbolic to only those variables that are actually used, and to batch many such tests into few instructions. Since this deferred testing can result in our erroneously executing instructions on symbolic data, we show how transactional memory can be leveraged to undo the effects of these erroneous computations.
- We detail the numerous optimizations necessary to realize the promise of this approach, in terms of achieving compelling performance improvements for some applications of symbolic execution. We show through microbenchmark tests where TASE outperforms modern alternatives, KLEE, S^2E, and QSYM. We also compare to contemporaneously developed symbolic execution tool SymCC [39].
- We show that TASE improves a specific defense using symbolic execution, namely behavioral verification [19], [16], to an extent that qualitatively improves how such a defense can be deployed on TLS traffic. Specifically, we show that TASE reduces the costs of this defense to permit its application *inline* for all but very latency-sensitive applications. In doing so, TASE enables preemptively protecting the server from exploits using this approach, versus its current ability to only detect malformed client messages alongside their processing by the server. We also demonstrate the flexibility of TASE by developing memory protections as an application of its techniques.

The rest of this paper is structured as follows. We discuss related work in Sec. II. We provide background and describe challenges that we must overcome to realize TASE in Sec. III, and present the design of TASE in Sec. IV. We discuss additional aspects of TASE's implementation in Sec. V. Sec. VI contains an evaluation of TASE for a symbolic execution-based application, and microbenchmarks. We discuss limitations of TASE in Sec. VII and conclude in Sec. VIII.

II. Related Work

We now outline prior work on symbolic execution systems and work using Intel's Transactional Synchronization Instructions.

A. Symbolic Execution Engines

Symbolic execution engines DART [25] and CUTE [43] represent some of the earliest modern attempts to mix concrete and symbolic execution [12]. Their approach, called concolic testing, analyzes a program by choosing an initial set of concrete input values V to a given program. The program is then executed with instrumentation to determine when control flow instructions are encountered, and constraints are accumulated at these branch locations in terms of their relation to the concrete inputs. After execution with input V terminates or is suspended, the constraints gathered from execution on V are analyzed to determine a new set of concrete inputs V' to guide execution down a different path. The process can be run repeatedly until all paths are explored, or until the tester wishes to cease path exploration. Although we prioritize native execution in TASE and mix concrete and symbolic execution, our approach differs from concolic execution in that we do not require entirely concrete inputs to drive symbolic execution. We also do not require re-execution of a program from a new set of concrete values to reach different execution paths, as we employ native forking (see Sec. IV-E) to explore different branches of program execution when control flow depends on the value of a symbolic variable. QSYM is another example of a concolic execution tool that requires re-execution to explore new execution paths [55]. Like TASE, QSYM incorporates optimizations to reduce the cost of interpretation. However, unlike TASE, QSYM sacrifices soundness to optimize its performance for fuzzing. Another recent tool, SymCC [39], implements concolic execution using compile-time instrumentation inserted into the LLVM IR of a program before machine code is generated.

Rather than using a program's native execution state as its primary representation, the KLEE symbolic execution engine [10] instead analyzes a program by interpreting its source

code translated to LLVM IR. KLEE is deeply optimized to minimize the cost of constraint solving by caching previous query results, applying normalization to constraints and queries to facilitate comparisons between expressions, and analyzing queries to determine subexpressions which may have already been solved. KLEE is also structured to explore multiple program paths within a single process. By doing so, KLEE is able to closely guide state exploration with heuristics chosen to prioritize code coverage or search for specific bugs or problematic behavior. KLEE also implements software based copy-on-write to more efficiently manage the symbolic states associated with different program paths.

EXE [11], Mayhem [13], and S^2E [18] are symbolic execution engines that use a program's native state as its principal representation. EXE analyzes a program by executing it natively and checking each use of a variable against a map that indicates if the variable is symbolic. Similarly, Mayhem uses dynamic taint analysis [38] to detect instruction blocks that touch symbolic data, while otherwise executing the program natively. EXE and Mayhem also use forking to explore multiple execution paths, i.e., forking the symbolic-execution process upon reaching a symbolic branch, to allow the parent and child to explore the two possibilities separately. S^2E uses QEMU and KLEE together to mix concrete and native execution, and is the system most similar to TASE. S^2E uses the virtualization and emulation tools within QEMU to perform symbolic execution across user space and kernel space boundaries [18]. S^2E also uses an emulated MMU that checks each byte during access in concrete execution mode to determine if control must transfer to the KLEE-based interpreter [17]. While we build on their techniques for sharing symbolic and concrete state, TASE is built to prioritize and optimize native execution using new transactional machine instructions and symbolic-state detection mechanisms detailed in Sec. IV. For detecting symbolic state, TASE does not solely rely on the bitmap lookup techniques used in EXE and S^2E, and TASE incurs no virtualization or dynamic binary translation overheads when executing code natively.

B. Intel TSX

Intel's Transactional Synchronization Instructions (TSX) were originally introduced to speed up concurrency in multithreaded applications [28, Ch. 16]. However, TSX instructions have been repurposed for security defenses (e.g., [45], [14]) and attacks [53], [22], as well. Similarly, TASE uses the transactions enabled by TSX in an unorthodox way. Specifically, TASE uses transactions to speculatively execute regions of code natively during symbolic execution, aborting the transaction if symbolic data is encountered. Key challenges for implementing this strategy are presented in Sec. III-B.

III. Background and Challenges

Our work to optimize symbolic execution for latency-sensitive applications required us to build on research from seemingly unrelated topics. In this section we briefly cover necessary background and key challenges that we address in TASE, pertaining to executing concrete operations natively but safely during symbolic execution (Sec. III-A) and leveraging Intel TSX in this context (Sec. III-B).

A. Concrete Operations in Symbolic Execution

Past works (e.g., [25], [11], [18], [13]) have recognized the significance of enabling native execution for entirely concrete computations in symbolic execution engines. However, the overwhelming amount of such concrete operations present in some of our target applications necessitate more aggressive optimizations in TASE. For example, in Chi et al.'s verification of OpenSSL traffic [16], which we explore as an application of TASE in Sec. VI-B, fewer than 2.7% of instructions executed operate on symbolic data, even after extensive protocol-specific optimizations to eliminate unnecessary concrete operations (described as the *optimized* configuration in Sec. VI-B1). To enable inline operation of this verifier, it is thus necessary that concrete operation be optimized as much as possible.

To do so, TASE speculatively executes regions of code natively within transactions, optimistically assuming that no operation in the transaction reads or overwrites symbolic values. Transactions are atomic, and if any operation in a transaction reads or overwrites a symbolic value, TASE must abort the transaction and resume execution within an interpreter—in our case, a modified version of the KLEE interpreter. After the transaction completes within the interpreter, TASE resumes native execution if possible.

Separating concrete and symbolic execution into different execution modes provided challenges for safely handling the symbolic expressions the interpreter produces. In particular, TASE tracks symbolic values by tainting them, specifically by augmenting KLEE's concrete/symbolic bitmaps with poison tainting and tracking. This required the design and verification of invariants to guarantee that the transition between concrete and symbolic execution does not unexpectedly overtaint or undertaint the program's execution with symbolic values, invaliding the resulting analysis. Moreover, because execution no longer occurs entirely within an interpreter, there is a risk that native execution might overwrite previously symbolic variables with concrete data with no indication to the interpreter, forcing us to adjust KLEE's data structures to prevent such updates.

B. Implementing Transactions with TSX

A key contribution of our work is the use of Intel Transactional Synchronization Instructions (TSX) to increase the speed of symbolic execution. We focus specifically on the use of the TSX Restricted Transactional Memory instructions `xbegin` and `xend`.

Intel's TSX instructions were originally released to provide a hardware-assisted tool for managing concurrency in a process. A thread thd may speculatively attempt to acquire a shared resource by using an `xbegin` prior to entering the critical section. `xbegin` starts a transaction in which any modifications to memory or registers made by thd are either entirely committed at the end of the transaction (signified by `xend`) or entirely discarded, at which time control for thd may transfer to a fallback path with simpler locking primitives (e.g., a spin lock). In other words, the transaction is atomic.

Should another thread thd' attempt to enter the critical section and modify the shared resource while thd is also altering the resource in the transaction, one or both of the transactions will abort and roll back [28, Ch. 16]. Transactions are rolled

back when conflicts over shared resources are detected between the read and write sets of thd and thd′, potentially allowing both threads to operate in the critical section simultaneously if thd and thd′ do not read or write the same shared data. Conflicts in the read/write sets of thd and thd′ are detected by the cache coherence protocol, and enabling concurrency with TSX can potentially outperform other locking methods which categorically prevent multiple threads from executing in the critical section concurrently, even if no conflicting memory accesses would have occurred [28, Ch. 16].

Intel's transactional execution instructions provide the basis for our speculative execution scheme. The application of the transactions to create a fast path, while conceptually simple, requires a large number of details to be addressed. First, as noted by Shih et al. [45], forcing a program to execute entirely within transactions introduces substantial challenges. Placing each basic block from the program within a single transaction introduces an overhead of roughly $8\times$ native execution, and transaction size is limited by cache size and associativity. Further complicating matters, transactions may abort due to asynchronous interrupts, are never guaranteed to commit, and must be carefully started and committed to avoid nesting.

Second, our speculative native execution scheme requires an efficient mechanism to abort transactions that encounter symbolic data. Ideally, individual bytes containing symbolic values could be marked as inaccessible by the OS (e.g., via page permissions) or a low-level hardware mechanism (e.g., via debug registers) so as to force any transaction accessing the byte to roll back. Unfortunately, the large granularity of page-level permissions and the scarcity of debug registers limit the effectiveness of these solutions. Another option is to inject instrumentation into the program to query a lookup table on each byte access (cf., [11]); however this approach incurs a performance penalty for additional read operations and compare operations, may clobber the FLAGS register depending on its implementation, and also impacts the number of operations that may be placed within a single transaction. Sec. IV contains our approach for overcoming these challenges.

IV. Design

In this section, we outline the design of TASE. We begin by describing the overall architecture of TASE, and follow with descriptions of the system's transactional execution; its poison checking scheme for detecting memory accesses of symbolic values; its method of interpretation; and its mechanisms for managing state exploration.

A. Structure of TASE

In TASE, we provide a symbolic execution system designed to rapidly symbolically execute user-space programs with small amounts of symbolic data. At its core, TASE provides a "fast path" and "slow path" for handling concrete and symbolic operations, respectively, as it executes an application (henceforth referred to as the *project*). Fig. 1 shows a simplified overview of these two primary components.

TASE requires C source code to execute a project, including source code for any C libraries the project will use. The "fast path" for native execution described earlier is an instrumented, binary x86 version of the project (and any

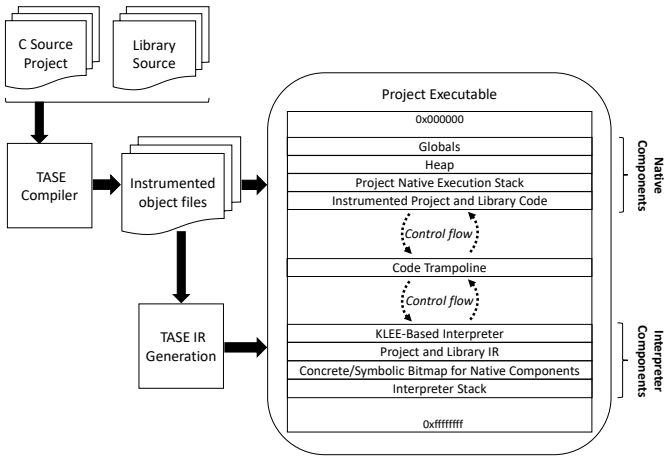


Fig. 1: High-level structure of TASE. TASE comprises components labeled "TASE Compiler" and "TASE IR Generation". TASE generates a project executable containing native and interpretable representations of the project source, and that switches between these representations through a code trampoline to which control flows after native execution of a project basic block and after interpretation of a project basic block that leaves no symbolic values in the emulated registers.

libraries it uses) produced by compiling the project's source code with our custom LLVM TASE compiler. Crucially, TASE executes within this instrumented native execution path as the rule rather than the exception. By instrumenting loads and stores and inserting jumps to a code trampoline (cf., [32], [45]) with transactional instructions around basic blocks, TASE enables speculative native execution. TASE uses a *poison* (or *sentinel*) value to mark bytes as containing symbolic values while executing the project. While executing code natively within a transaction, values read and overwritten are recorded and checked en masse with SIMD instructions at the end of a transaction. If the poison value was read or overwritten, the transaction is aborted and all state changes performed during the transaction are undone; details are provided in Sec. IV-B.

If TASE is unable to complete a transaction natively, control transfers via a context switch from the trampoline to the "slow path", our KLEE-based interpreter. The interpreter is responsible for executing the target binary until another transactional entry point is reached, at which time the target's execution might begin again concretely. KLEE executes by interpreting LLVM bitcode exclusively, whereas TASE switches between instrumented native execution to efficiently execute concrete operations and KLEE-style interpretation to handle symbolic operations. Although KLEE normally runs on IR generated directly from C code [10], TASE uses KLEE to interpret through IR generated to represent x86 semantics; more details on the mechanics of TASE's interpretation are provided in Sec. IV-D. Additionally, TASE differs from KLEE in its use of processes (cf., [11]) to represent execution states (see Sec. IV-E).

Context switching between the interpreter and native execution in TASE closely resembles that in S^2E [18]. The interpreter and native execution share a common address space, and a context switch from native execution to the interpreter

occurs by snapshotting the current state of the general purpose registers (GPRs). The interpreter then uses this snapshot to model each x86 instruction's effects on main memory and a simulated copy of the GPRs, which is restored for concrete execution after a transactional boundary is reached and symbolic values no longer reside within the GPRs.

Symbolic data—including values used for altering control flow—are exclusively handled by the interpreter, which may also fork state to explore new execution paths. Our forking mechanism uses the native Unix `fork` system call to explore execution paths, similar to the techniques used in EXE [11]; we include more details in Sec. IV-E. We discuss the mechanisms for detecting usage of symbolic data during native execution in Sec. IV-C.

B. Transactional Execution

To mitigate the cost of interpreting instructions with concrete operands, TASE instead executes these instructions natively within TSX transactions. Our strategy is to speculatively execute the target program natively for as many transactions as possible, and abort a transaction if an "unsafe" operation occurs that requires special handling of symbolic data via interpretation. TASE requires access to source code, and emits instrumented machine code for the program along with the symbolic execution components and interpreter in a single executable.

In a previous work that executed software in transactions for a different purpose, T-SGX [45] uses the Clang LLVM back-end to conservatively estimate the read and write sets of instructions and cache-way usage at compile time to efficiently group together a large number of instructions in a single TSX transaction. Their technique helps to maximize the number of instructions in a transaction to amortize the overhead required for setting up and committing or rolling back a transaction. Unlike T-SGX, TASE does not statically determine the number of instructions to place in a transaction. Our evaluation of OpenSSL verification (see Sec. VI-B) revealed the need to efficiently instrument code that frequently included variable-sized loops and function pointers, both of which make effective compile-time instrumentation challenging. Using a custom Clang LLVM back-end, TASE injects trampoline jumps around basic blocks and dynamically determines the boundaries for closing and opening a transaction at runtime.

For any transaction, let its *stride* denote the number of basic blocks attempted within the transaction. In our present implementation, we currently use a transactional batching policy in which the stride of each transaction, by default, is set to a constant s_{max}; in our evaluation in Sec. VI, s_{max} is set to 16. If a transaction tx aborts, then one possibility would be to trap to the interpreter and simply interpret through the whole aborted transaction. However, a more refined approach that leverages the *reason* for the abort can optimize execution considerably.

If tx aborts due to reading or overwriting a poisoned memory location (see Sec. IV-C), then tx is aborted using an `xabort` instruction. This instruction permits information about the abort to be conveyed to the abort handler in a register. We use this facility to convey the number c of basic blocks that were successfully executed in the transaction (without detecting poison) before the one where poison was encountered, which is tracked in a counter updated by the trampoline. In this case, TASE attempts another transaction tx' beginning at the same place as tx, but with a stride of c. If tx' completes successfully, then the interpreter is invoked to interpret through the next basic block (where poison is known to appear), and native execution is resumed afterward, if possible.

If tx aborts for another reason, then it is generally necessary to interpret through the basic block where the abort occurred (see [28, Sec. 16.3.8.2]). For example, if tx aborted due to triggering a page fault, then it will likely trigger the page fault again if retried in full [30]. TASE thus attempts to natively execute as many of the basic blocks in tx as possible while incurring few transaction aborts, before leveraging the interpreter to interpret through the basic block that caused tx to abort. TASE does this using the following logic (inspired by binary search), assuming the stride of tx is s_{max}:

1) $s \leftarrow s_{\mathsf{max}}/2$.
2) While $s \geq s_{\mathsf{min}}$ do:
   a) Attempt a transaction tx' of s basic blocks.
   b) $s \leftarrow s/2$ (regardless of whether tx' aborted and, if so, the reason for the abort).
3) Trap to the interpreter and have it interpret through s_{min} basic blocks.

After step 3, TASE resumes native execution with its default stride of s_{max}. Note that each tx' of stride s in step 2a will either advance the program counter past the s blocks attempted if tx' does not abort, or will leave the program counter unchanged if tx' aborts. The logic above attempts to ensure that if the condition that caused tx to abort is persistent (e.g., a page fault incurred during a particular basic block), then the troublesome basic block is interpreted in step 3.

C. Poison Checking

To prevent native transactions from interacting with symbolic information in an "unsafe" way, we implement a poison checking scheme. One such "unsafe" interaction we prohibit is the loading of a symbolic variable into a register; this allows us to assume that arithmetic performed within the general purpose registers cannot access symbolic values and therefore needs no additional instrumentation.

Broadly speaking, on a byte-level basis each memory location that contains a symbolic value or expression is "poisoned" with a reserved numeric value. Reads and writes within a native transaction are instrumented at compile time to store the values read and overwritten in reserved SIMD and general purpose registers. At the end of each basic block and prior to a transaction's commitment, the values in the registers are tested in bulk to determine if a poison value was read or overwritten. If a poison value is found in the SIMD registers reserved for instrumentation, the transaction is aborted.

To make this scheme sound and efficient, several implementation refinements were required. First, we implement poisoning on an aligned two-byte basis. Byte-level poisoning would potentially result in a large number of "false positives" in which a native transaction reads a value concretely that, by coincidence, matches the poison value. Consequently, if any

single byte b needs to be marked as symbolic by the interpreter, we poison the 2-byte-aligned buffer B containing b. If B also contains a concrete byte b', then we mark b' as symbolic as well, but with a constraint that b' equals its concrete value prior to B's poisoning. Of course, the risk of false positives could be reduced further by poisoning 4-byte buffers, though we have found 2-byte poisoning to suffice so far.

Second, to prevent natively executing instructions that read from or write to memory locations containing symbolic data, we copy reads and memory values prior to writes to our reserved SIMD and general purpose registers during native execution. These read and overwritten values are later checked at the end of the basic block to determine if an instruction could have operated on symbolic data. If so, the transaction rolls back and the interpreter handles the transaction, referencing its internal bitmap indicator to determine if memory operands are concrete or symbolic. For example, suppose some instruction instr in basic block blk performs a load or store to a symbolic address (i.e., a symbolic pointer) during native execution. As noted earlier, TASE remains in the interpreter while symbolic data remains in its simulated registers, so the registers must be entirely concrete at the beginning of blk. Some instruction instr′ (possibly instr itself) must read from memory (explicitly or as a side-effect) the poisoned symbolic data used as an address by instr. When instr′ is executed, TASE's instrumentation copies the poison tag associated with symbolic data to its reserved SIMD registers for batched checking at the end of the basic block, at which time the transaction will abort. In this example, even if instructions after instr′ in the basic block perform memory operations using malformed addresses impacted by the poison tag (e.g., calculating offsets from the poison), our checks at the end of the basic block discard the changes (or the transaction will roll back via segmentation fault if the malformed address is for an unmapped page). In other words, when the first poison value is read or overwritten within a basic block and stashed away for batched poison checking later, regardless of subsequent instructions, the transaction containing the entire basic block is destined to abort. Compared to instruction-by-instruction instrumentation, batching poison checks at the end of the basic block in SIMD registers reduces the total number of instructions required to perform the poison checks, and simplifies the process of verifying that instrumentation checks do not clobber the FLAGS register.

Third, to ensure that control flow within a transaction containing one or more instructions operating on poisoned data reaches the SIMD checks and aborts, we add additional instrumentation before indirect control flow instructions which allow jumps to arbitrary destinations. Without such a safeguard, our belated poison checking scheme could allow poison-dependent indirect control flow arithmetic (e.g., jump table calculations and function pointers) to erroneously transfer control to destination addresses computed with the poison sentinel value, thereby circumventing the poison checking logic. Specifically, we add an additional trampoline jump to the poison checking logic before instructions that jump to an operand address (e.g., `call %rax`, `jmp %rax`), effectively placing the instructions in a separate basic block and preventing their execution if the operand is symbolic; as stated earlier, control flow between basic blocks traps to our interpreter if symbolic taint enters a register. Similarly, if the indirect control flow instruction performs a jump to an address stored in memory pointed to by its operands (e.g., `ret`), we inject an additional poison check for the destination address and a jump to the batched poison-checking logic before the instruction is executed. Fortunately, assuming access to the source code for target applications in TASE and restricting our custom compiler's instruction selection simplifies the task of instrumenting indirect control flow instructions.

The above design points help to ensure that native execution does not interact with symbolic values in any way, including "clobbering" writes to symbolic variables that would, in effect, concretize them without notifying the interpreter. Our "in-place" poison-checking scheme along with KLEE's symbolic bitmap indicator provides a mechanism for accomplishing this.

The design choices for our poisoning scheme were made to minimize the cost of instrumentation. Whenever possible, our instrumentation to save values read from or overwritten in memory is inserted into the target code to prevent any additional reads from or writes to memory. With the help of alignment guarantees from the compiler, many instructions reading data larger than a byte can be instrumented by moving the data read from a general purpose register to a reserved register, where it is later checked en masse with other data values. Taint trackers such as Minemu [6] use similar techniques to reserve SIMD registers for instrumentation purposes. Although TASE does not currently support native execution or interpretation of SIMD instructions (i.e., SIMD instructions are currently only executed as instrumentation), this limitation is not fundamental; we plan to extend TASE in future work to emit SIMD instructions as part of the program being symbolically executed, as our poisoning scheme only requires three SIMD registers to be reserved.

D. Interpretation

Should a native transaction encounter symbolic data, control flow in TASE transfers to a KLEE-based interpreter. Given the representation of the project as a set of x86 registers and an address space, the interpreter executes instructions until a transactional entry point is reached (i.e., an instruction corresponding to the code trampoline) and the registers contain no symbolic data. The interpreter tracks symbolic data on a per-byte basis.

Invoking the interpreter requires saving a snapshot of the GPRs as they appeared at the end of the last successful transaction; crucially, TASE does not require that main memory is snapshotted or copied during the context switch. In the interpreter, reads and writes to main memory are performed directly on the addresses being read from or written to. However, changes to the simulated x86 registers in the interpreter must be faithfully tracked so that native execution can be resumed by a context switch after interpretation has completed. As KLEE interprets LLVM IR, we provide LLVM IR representations of each x86 machine instruction within a given transaction to preserve the semantics of the program and produce a system state that may be restored for native execution; see Fig. 2a and Fig. 2b for a simplified example of how the x86 instruction `pop %r9` is modeled, and Sec. V-A for details on further optimizations. In order to avoid directly

```
1 void interp_pop_r9 (greg_t* gregs) {
2   gregs[REG_R9] = *(greg_t*)gregs[REG_RSP];
3   gregs[REG_RSP] = gregs[REG_RSP] + 8;
4   gregs[REG_RIP] = gregs[REG_RIP] + 2;
5 }
```

(a) C model

```
1 define void @interp_pop_r9(i64* nocapture %gregs) #1 {
2   %1 = getelementptr inbounds i64* %gregs, i64 15
3   %2 = load i64* %1
4   %3 = inttoptr i64 %2 to i64*
5   %4 = load i64* %3
6   %5 = getelementptr inbounds i64* %gregs, i64 1
7   store i64 %4, i64* %5
8   %6 = add nsw i64 %2, 8
9   store i64 %6, i64* %1
10  %7 = getelementptr inbounds i64* %gregs, i64 16
11  %8 = load i64* %7
12  %9 = add nsw i64 %8, 2
13  store i64 %9, i64* %7
14  ret void
15 }
```

(b) LLVM IR model

Fig. 2: Simplified models for interpreting `pop %r9`

writing LLVM IR, our method for producing an LLVM IR model for each x86 instruction is to write the state changes performed by the instruction in C (as in Fig. 2a), and use Clang (https://clang.llvm.org) to emit LLVM IR (as in Fig. 2b). Note that the interpretation of `pop %r9` in Fig. 2a is modeled as the execution of a function. The function takes a context containing a set of simulated general registers (`greg_t * gregs`), copies the value in main memory pointed to by the simulated stack pointer to the interpreter's simulated register `%r9` (line 2 in Fig. 2a), and increments the simulated stack and instruction pointers (lines 3–4 in Fig. 2a).

Equivalent LLVM IR is provided in Fig. 2b. Note that the offsets of the stack pointer, `%r9`, and the instruction pointer in the `greg_t * gregs` struct are 15, 1, and 16, respectively. The instructions on lines 2–5 retrieve the value of our simulated stack pointer into temporary variable `%2` and load the value at that address into `%4`, and lines 6–7 copy this value into the interpreter's model of `%r9`. In line 5, our interpreter directly reads from TASE's virtual memory located at the address specified by our model of `%rsp` in the `greg_t * gregs` struct; because of this, context switches between the interpreter and native execution do not require expensive copy operations for memory other than the saving and restoring of our simulated registers. The instructions on lines 8–9 increment our simulated stack pointer and correspond to line 3 in Fig. 2a. Similarly, lines 10–13 increment the simulated instruction pointer to point to the next opcode, as in line 4 of Fig. 2a.

While the example described above and pictured in Fig. 2 is for a single instruction, TASE instead generates interpretable models for entire basic blocks of the original project. We elaborate further in Sec. V-A.

E. State Management

In addition to managing the transition between native execution and interpretation, TASE must also handle the exploration of a potentially large number of execution paths. Handling this "state explosion" problem is a crucial aspect of symbolic execution, and has been a primary concern of many papers [12], [10], [13], [17].

In TASE, multiple execution paths are explored in parallel by using a native forking mechanism. Unlike other systems that explore multiple execution states within a single address space, TASE is unable to handle multiple execution states concurrently within a single address space. Attempting to explore states concurrently with multiple threads on one address space could cause unintended transactional aborts when threads access a common memory address.

Whenever the target program encounters a control flow instruction (e.g., a `jmp` or branch) that depends on a symbolic variable, execution must revert to the interpreter. After the interpreter takes control, execution states are created corresponding to the different possible destinations of the control flow instruction, and the `fork` system call is invoked. The resulting two processes extend the current execution in cases that the branch condition is true or false, respectively. We address indirect control-flow transfers dependent on a symbolic variable by producing an execution state for each possible destination. EXE [11] uses a similar mechanism to handle state exploration, and in both TASE and EXE this approach provides the benefit of hardware-based copy-on-write to mitigate the cost of creating new processes. Both EXE and TASE also have at least some cases in which state exploration and path prioritization require child processes to halt and wait for a central state management process to authorize further execution. This potentially introduces bottlenecks when many child processes are exploring a large state space; however the centralization of state management in a single process helps to prevent "fork bombing" issues in which the machine hosting TASE is overwhelmed with too many processes.

Forcing each process following a `fork` to signal back to the central management process allows a variety of search heuristics to be implemented by the central coordinator. We intend to explore the use of simple heuristics, such as breadth-first and depth-first search, as well as ones tailored to particular applications. For example, in prior research on client behavior verification, Cochran et al. [19] leveraged the next message inbound from the client to prioritize the order in which paths were explored to identify a path consistent with that message having been sent next by the client. This prioritization was based on data collected from the client program during a training phase. In this approach, when a path search reaches a symbolic branch, the central coordinator determines which of the currently paused processes—i.e., either the two resulting from this `fork`, or another one—is on a path that is "closest" to one that, in training, could typically be used to "explain" the latest message received from the client. That process would then be signaled to continue its search until reaching the next symbolic branch. Of course, this prioritization is only an example strategy, and we intend to explore others, as well.

V. Implementation

In this section we briefly discuss implementation details of TASE.

A. IR Generation

Like other symbolic execution engines, TASE requires an intermediate representation of code to perform symbolic execution. Specifically, TASE uses LLVM IR to model each x86 instruction that potentially touches symbolic data, as discussed in Sec. IV-D.

Crucially, unlike some other symbolic execution tools, TASE requires access to source code, from which TASE produces an instrumented executable using a custom compiler. Controlling the compiler allows us to selectively limit the pool of instructions available to the LLVM backend's code-generation algorithms. This drastically simplifies the laborious task of producing IR models for x86 instructions, at the cost of requiring source code.

Additionally, we use information provided by the LLVM backend during compilation to record FLAGS-register liveness information around basic blocks, which we use to periodically kill the FLAGS register. This benefits our execution in TASE because it reduces the overall amount of symbolic data the interpreter must handle, and, in certain situations, allows the interpreter to more quickly produce a fully concrete copy of its simulated GPRs needed to return to native execution.

Because execution within a basic block in TASE must occur either entirely in the interpreter or natively for the duration of the basic block, we employ an additional optimization to speed up interpretation. We "batch" the IR for all x86 instructions in a basic block together and invoke the interpreter to interpret the whole basic block at once, rather than doing so per instruction within the basic block. In practice, we observe that this optimization reduces the total size of the LLVM interpretation bitcode by a factor of roughly three. Assuming access to source and control over the compiler also helps here; by disabling the selection of instructions that modify certain flags bits (e.g., the direction flag used by string-manipulation instructions), the overall size of the IR is reduced and more opportunities to omit redundant flags computations appear. Moreover, we found that reducing instruction selection based on flags usage offered opportunities to completely kill flags in certain cases after control flow instructions were used, reducing the likelihood of expressions "snowballing" together due to flags computations being continuously OR'd together.

Finally, we structured our C models of x86 instructions to more effectively use the compiler's aliasing optimizations. For example, using the "restrict" keyword before accessing our simulated register file or simply using local variables (rather than pointer access, as used in Fig. 2) helped the compiler to optimize as if the simulated registers and simulated memory were separate address spaces, thus reducing the size the LLVM IR models of the x86 instructions.

B. Forking and Path Exploration

As noted in Sec. IV-E, we employ a native Unix `fork` call to explore multiple execution states in TASE when execution encounters a symbolic branch. Execution in TASE begins with a central "manager" process forking off a child process to begin path exploration of the project's code. The manager uses signal-based job-control mechanisms, shared memory, and system-level semaphores to steer and control execution through different branches as a pre-defined maximum number of worker processes execute in parallel. If a worker encounters a symbolic branch, it halts execution until the manager process determines what course of action to take.

Native forking in TASE benefits from hardware-based copy-on-write, but still incurs overhead; among other things, the Linux kernel copies the parent process' page table entries for the child [24]. To reduce this cost, our experiments in TASE use the Linux transparent huge pages feature to reduce the size of page table mappings without explicitly modifying the applications. The daemon used by the kernel to coalesce small (4KB) pages into huge (2MB) pages periodically runs at a predefined interval; we experimentally determined that 10ms appeared roughly optimal for our behavioral verification application in Sec. VI-B.

C. Transaction Sizing

As discussed in Sec. IV-B, the stride of a transaction in TASE is set to a constant s_{max} by default; after executing s_{max} basic blocks, the transaction will be closed. A value s_{max} that is too small will hurt performance by closing transactions more frequently than necessary, whereas a value that is too large can incur a substantial performance penalty when a transaction aborts, since all the work it performed will be thrown away. To maximize performance, s_{max} would ideally be tuned per project and per platform, since the size of the L1 data cache limits the amount of data that a transaction can read or write and since the frequency at which symbolic data is accessed may vary depending on the application. In the future, we plan to explore dynamically adjusting s_{max} based on runtime conditions, as well. For the purposes of our evaluation in Sec. VI, we simply set $s_{\mathsf{max}} = 16$.

Because the basic block is the smallest granularity at which transaction size can be controlled in TASE, it is also necessary that basic blocks be limited to a maximum size. In our present implementation, basic blocks are limited to 50 instructions. Here again, the limit of 50 was chosen experimentally; we plan to explore methods in future work to automatically tune this constant.

VI. Evaluation

In this section we measure TASE's performance. We first detail TASE's performance in a series of microbenchmarks in Sec. VI-A, and then we consider an application of symbolic execution to validating the messaging behavior of a software client in Sec. VI-B. Finally, we explore application of TASE's techniques to memory protection in Sec. VI-C. All performance experiments described in this section were conducted on a computer with a 3.5GHz Intel Xeon CPU E3-1240 v5 processor and 64GB of RAM. All tools in the evaluation were either Dockerized,[2] or, if not Dockerized, ran directly on Ubuntu 16.04.7.

[2]All Dockerized tools were executed in a fully-privileged container (i.e., with the "–privileged" flag). Surprisingly, we observed that performance degraded by up to a factor of 2 when containers were created with default permissions.

Test	TASE	SymCC	S^{2}E	QSYM	KLEE
BigNum add	13.15×	11.40×	42.79×	903.63×	2403.53×
sha256	8.96×	13.95×	18.15×	2239.19×	1738.23×
md5sum	12.27×	17.45×	71.94×	1904.99×	7208.67×
cksum	2.83×	7.23×	10.25×	691.18×	1137.48×
tsort	15.11×	7.51×	35.56×	1073.89×	>20,000×
factor	7.16×	4.41×	30.32×	1131.95×	1070.42×

TABLE I: Concrete computation costs relative to native execution, averaged over five runs; relative standard deviations were $< 5.3\%$. BigNum addition was performed byte-by-byte on two 10MB integers. Hashes were computed on a 44MB file. tsort was run on a file with 500,000 edges. factor was run on a 38-digit number with five prime factors.

A. Microbenchmarks

In this section we report the results of various microbenchmarks to compare TASE to alternatives when executing on mostly concrete workloads—the contexts for which TASE was designed. Our first microbenchmark evaluations compared TASE to native execution and execution by S^{2}E, QSYM, SymCC, and KLEE,[3] for six programs: the first added two concrete 10MB integers byte-by-byte with a one-byte carry; sha256[4], md5sum[5], and cksum[6] were each applied to a concrete 44MB file; tsort[7] was run on a file with 500,000 edges; and factor[8] was run on a 38-digit number with five prime factors. These programs were compiled using Clang 7.1.0 with O2 optimization for the native, S^{2}E, and QSYM targets, and with Clang 9.0.1 with O2 optimization for KLEE (as 9.0.1 was the Clang version included with KLEE). SymCC's custom compiler was run with O2 optimization. In contrast, TASE supports only a limited version of O1 optimization at the time of this writing. The results of these executions are shown in Table I. TASE overheads ranged between ≈3–15× native execution on these benchmark programs. SymCC overheads ranged from ≈4–17× native execution. S^{2}E was ≈10–72× slower than native, and QSYM and KLEE incurred overheads of ≈691–2239× and from ≈1070× to over 20,000× native execution, respectively.

TASE is tailored to executing projects with small amounts of symbolic data, and so increasing the amount of symbolic data does impact its performance. Fig. 3 shows the performance of byte-by-byte BigNum addition using the same code represented in Table I (but only 50KB operands), but with a byte at a varying index marked symbolic. Once this byte is encountered, the carry byte becomes symbolic and remains so for the rest of the computation; as such, the bytes of the sum tainted by the symbolic carry byte are symbolic, as well.

The location of this symbolic data did not affect the performance of the BigNum addition in KLEE (Fig. 3), since KLEE interprets all project instructions. The performance of the BigNum addition in TASE, however, decayed as the first symbolic byte was encountered earlier in the computation; only once ≈ 75% of the BigNum addition was performed concretely does TASE outperform KLEE. The primary reason for the loss of performance for larger amounts of symbolic data is that TASE interprets substantially more LLVM IR instructions to model a restorable context for native execution (see Sec. VII-C). In particular, TASE executed almost 8× as many IR instructions as KLEE in the BigNum addition test when the index of the first symbolic byte was set to zero. S^{2}E exhibited similar trends as TASE, eventually becoming faster than KLEE; however, its performance was much worse than TASE in this test and became faster than KLEE only once $> 95\%$ of the BigNum addition was concrete. Even with no symbolic computation whatsoever (i.e., with 0% symbolic input in Fig. 3), QSYM was at least 7× slower than both S^{2}E and TASE, and only about 25% faster than KLEE, although QSYM outperformed S^{2}E considerably after more than 4% of the total computation in the microbenchmark required the manipulation of symbolic data. Both TASE and SymCC performed similarly in Fig. 3.

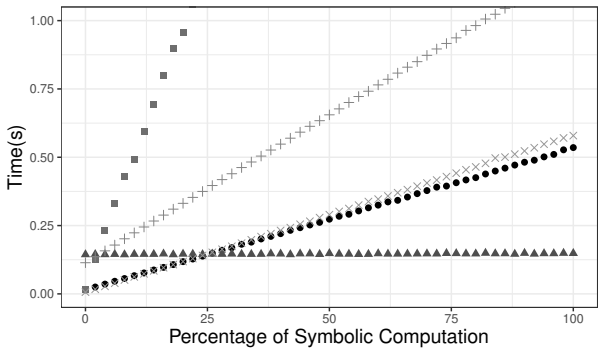


Fig. 3: Average time to add two 50KB integers vs. amount of symbolic computation, for TASE (●), KLEE (▲), S^{2}E (■), SymCC (×), and QSYM (+). Each point is an average over five runs; relative standard deviation was $< 13.7\%$. Initialization and setup costs were removed, though there were none for TASE. Symbolic data was inserted into one of the two arrays prior to the ripple addition at a variable index indicated by the percentage on the x-axis.

Based on the concrete overheads in Fig. 3 and Table I, and the Pin instrumentation code from the QSYM repository, we believe the slowdown on concrete workloads within QSYM is primarily due to its insertion of instrumentation functions before each machine instruction of the target. Although more lightweight than the Pin instrumentation in QSYM, SymCC's compile-time instrumentation resulted in the insertion of checks within the IR for some of our target programs, increasing the total size of the bitcode (e.g., increasing the total number of instructions in the IR for the BigNum microbenchmark by approximately 3×).

[3]We used Dockerized QSYM from February 10, 2020 (https://github.com/sslab-gatech/qsym); Dockerized KLEE from December 23, 2020 (https://klee.github.io/docker/); Dockerized SymCC from September 6, 2020 (https://github.com/eurecom-s3/symcc); and S^{2}E retrieved on July 11, 2019 (https://github.com/s2e/s2e-env.git).

[4]https://github.com/coreutils/gnulib/blob/master/lib/sha256.c

[5]https://github.com/kfl/mosml/blob/master/src/runtime/md5sum.c

[6]https://github.com/coreutils/coreutils/blob/master/src/cksum.c

[7]https://github.com/coreutils/coreutils/blob/master/src/tsort.c

[8]https://github.com/coreutils/coreutils/blob/master/src/factor.c

B. Client Verification with TASE

The second evaluation for TASE that we report is its use in verifying client behavior in client-server protocols. Numerous server exploits take the form of messages to a server that no legitimate or sanctioned client would send. To detect such messages in this approach, a server-side *verifier* symbolically executes the claimed client software (with unknown client-side inputs marked symbolic) to determine whether the message sequence received from the client is possible given the client software and messages sent to it (e.g., [5], [19]).

This defensive strategy benefits from its generality—it needs no foreknowledge of a vulnerability or exploit to detect an attack—and soundness, in the sense that if it accepts a sequence of messages, then there are inputs that could have caused the claimed client to produce that sequence. However, it also has limitations that restrict the contexts in which it can be applied. First, it requires a claim of the software executed by the client. This claim could be explicit via a user-agent string (as in HTTP, SIP, or NNTP) or an attested load-time measurement from a hardware trusted-computing platform (see Maene et al. [33] for a survey) or host-based monitor. Alternatively, this claim could be inferred from the client's behavior (e.g., [1]) or simply because the client was previously provisioned with software known to the verifier (e.g., as an IoT device might be). Second, the verifier must be able to obtain the claimed client software to symbolically execute—source code if the symbolic execution engine requires it.

The primary focus of this evaluation is a third challenge faced by this defensive approach: to be used as an inline defense, it requires symbolic execution to keep pace with the arrival of messages from the client. Thus, the latency of symbolic execution is critical in determining whether this defense can be used inline to *prevent* exploits, or whether it can only be run alongside server processing and thereby *detect* exploits shortly after they occur. We show not only that TASE significantly improves performance over a state-of-the-art codebase for conducting behavioral verification of a TLS client, but that it does so to an extent that permits this defense to reside on the critical path of message processing for all but the most latency-sensitive TLS applications.

The specific codebase to which we compare here is that due to Chi et al. [16], who instantiated this general approach for TLS. To determine whether or not a client message could have originated from an unmodified client TLS implementation, Chi et al. detail a technique for symbolically executing OpenSSL's `s_client` and then solving to determine whether there exist inputs that could have caused that implementation to produce the message sequence received. A message sequence for which no inputs can be found to produce it indicates that the message sequence is inconsistent with `s_client` and so might represent an exploit, and indeed, Chi et al. observed that numerous notable TLS exploits (e.g., Heartbleed, CVE-2014-0160) are of a form that would be caught by this technique. The Chi et al. framework is an extension of similar tools (e.g., [5], [19]) adapted specifically for cryptographic protocols like TLS: it leverages knowledge of the TLS session key and symbolically executes the client in multiple passes, skipping specified *prohibitive functions* (the AES block cipher and hash functions) until constraints generated from observed client-to-server messages could fully concretize their inputs. Below, we refer to the tool built by Chi et al. as CliVer (for simply "client verification").

The only changes we made to the CliVer tool for this evaluation was to implement the following two optimizations for it, to make the comparison to TASE fairer since TASE incorporates analogous optimizations. First, we changed how CliVer models the `select` system call, so that its return value indicates that `stdout` is always available (versus being symbolic). `s_client` writes the application payload received from the server to `stdout`, and so blocks if `stdout` is unavailable. As such, this change has no effect on the message sequence that could be received from `s_client`; i.e., any message sequence received in an execution where `stdout` becomes unavailable is a prefix of a sequence that could be received in an execution where it remains available throughout. This change does, however, relieve CliVer from needing to explore the execution path in `s_client` where `stdout` is unavailable, saving it the expense of doing so.

Second, when CliVer is seeking to verify message i from the client and reaches a send point when symbolically executing `s_client`, it must create and solve constraints reflecting message i and the path executed to reach that send point. This produces an unusually large number of relatively simple equality constraints (i.e., one constraint per each byte of the message), many of which contain a large number of XOR operations due to the choice of cipher suite. To more efficiently move the constraint information between the interpreter and its solver, we alter the behavior of KLEE's independent constraint solver to send all constraints en masse rather than one-by-one. Moreover, though the SAT solver we use supports XOR expressions [48], we found it much more efficient to rewrite these expressions to remove XORs before sending them to the solver. This optimization improves the performance of CliVer considerably, and we leverage it in TASE, as well.

1) Experiment setup: Our evaluation used the same TLS 1.2 dataset used by Chi et al. [16]. This dataset includes benign traffic captured by `tcpdump` during a Gmail browsing session, and maliciously crafted Heartbleed packets to simulate CVE-2014-0160. The Gmail data set was generated by sending and receiving emails with attachments in Firefox over a span of approximately 3 minutes, and included 21 independent, concurrent TLS sessions for a total of 3.8MB of data. For reference, a plot of the time during which each of the 21 TLS connections was active is shown in Fig. 4. As shown there, a large majority of connections were active for nearly the entire duration of the Gmail session, though a few were much shorter.

We compared TASE with CliVer in two configurations. The first presumes minimal protocol-specific knowledge or thus adaptation by the party deploying the verifier. In this *basic* configuration, each tool was provided a specification of the same prohibitive functions, but otherwise the tool operated on the Gmail trace unmodified. Even in this configuration, however, we provided CliVer with native implementations of these prohibitive functions, so that even once their inputs had been concretized, they would be executed natively (versus being interpreted), thus rendering our comparison conservative. The second, *optimized* configuration incorporated a range of protocol-specific optimizations. In particular, after the TLS 1.2 handshake, client-to-server and server-to-client messages are independent of one another, and so server-to-client messages

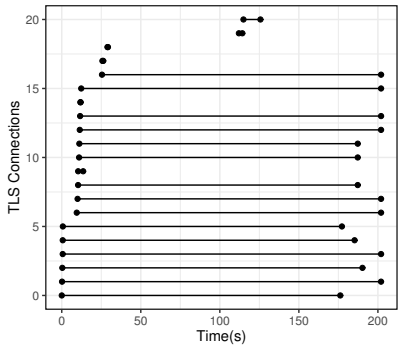


Fig. 4: The durations of the 21 TLS connections involved in the Gmail trace described in Sec. VI-B, ordered bottom-to-top according to the TLS connection initiation. Note that some connections are so brief that their beginning and ending markers overlap.

were ignored when verifying the client-to-server messages. In addition, certificate verification was elided, since the verifier, being deployed to protect the server, trusts the server to send a valid certificate chain.

2) Results: When used as an inline defense against malicious traffic, the speed to reach a true detection is arguably a secondary concern; the delay imposed on attack traffic might be viewed more as a benefit than a detriment. Nevertheless, we used synthetic Heartbleed packets to confirm that TASE could determine these packets were not consistent with the OpenSSL TLS client in only 150ms from the initiation of the connection (i.e., including the TLS handshake).

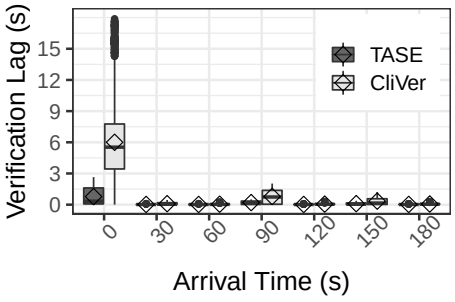
More critical is the delay that TASE would impose on legitimate traffic. Here we report the *cost* and *lag* of verification as defined by Chi et al. For the i-th message in a TLS session, $cost(i)$ is the processing time required to verify message i beginning from the symbolic state produced from verifying through message $i-1$. $lag(i)$ is the delay between the receipt of message i and when its verification completes. Note that $lag(i) \geq cost(i)$, and $lag(i) > cost(i)$ if when message i is received by the verifier, the verification of message $i-1$ is not yet complete (and so verifying message i cannot yet begin).

Table II gives coarse statistics for the cost and lag of verifying all 21 TLS sessions in the Gmail trace, in the *basic* (left) and *optimized* (right) configurations. Interestingly, the median costs for TASE and CliVer were very similar, but the median lag for CliVer was $27\times$ larger (in the *optimized* configuration) than it was for TASE. The cause was the messages that were most costly to verify, with costs in CliVer more than $14\times$ that in TASE in the *optimized* configuration (and roughly $29\times$ that in TASE in the *basic* configuration). These greater costs caused the lag to accumulate at various points in the trace, inducing an average CliVer lag on the *optimized* configuration of > 1s and a maximum lag of > 4s. In contrast, the TASE lag averaged only ≈ 0.1s and incurred a maximum lag for any message of $\approx$ .9s. For a driving application like Gmail that is paced by human activity, these lags may well be small enough to support the use of TASE as an inline defense.

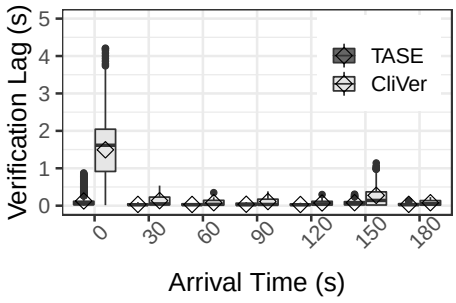
Configuration	*basic*		*optimized*	
System	TASE	CliVer	TASE	CliVer
Average cost	0.005s	0.033s	0.016s	0.071s
Median cost	0.003s	0.006s	0.013s	0.018s
Max cost	0.072s	2.116s	0.088s	1.274s
Average lag	0.614s	4.654s	0.106s	1.138s
Median lag	0.188s	4.721s	0.045s	1.195s
Max lag	2.648s	17.885s	0.876s	4.211s

TABLE II: Statistics for verification of benign Gmail traces

A temporal view of the lag is pictured in Fig. 5, which shows the distribution of lag across all 21 TLS sessions with messages binned according to their arrival times, where $arr(i)$ denotes the arrival time of message i; for example, the first bin contains the messages that arrived within the first 30s of each of the 21 TLS sessions. Arrival time is measured relative to the start of the individual TLS session. Within each bin, a box-and-whisker plot shows the first, second, and third quartiles, with the whiskers extended to $1.5\times$ the interquartile range. The average is shown as a diamond, and outliers appear as individual points.



(a) *basic* configuration

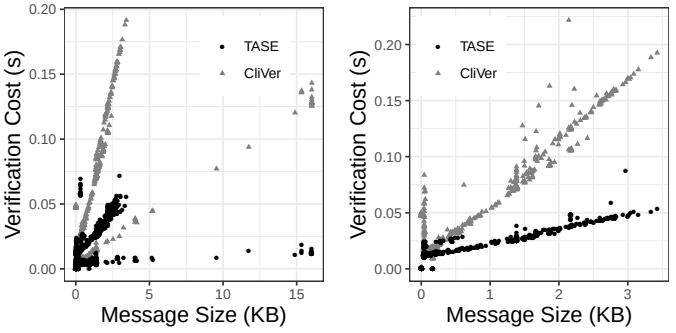


(b) *optimized* configuration

Fig. 5: Verification lag for 21 TLS sessions in the Gmail trace, each verified in isolation. The box plot at arrival time t includes $\{lag(i) : t \leq arr(i) < t + 30\text{s}\}$ across all 21 TLS sessions. Fig. 5a shows lag in verifying the Gmail traces using TASE and CliVer in a basic deployment without protocol-specific optimizations, and Fig. 5b shows lag in a configuration with optimizations leveraging protocol knowledge; see Sec. VI-B1.

The lags for the *basic* deployment lacking protocol-specific optimizations are shown in Fig. 5a, and the lags for the *optimized* deployment leveraging protocol-specific optimizations are shown in Fig. 5b. Both TASE and CliVer suffered lag in the first 30s of each connection, though TASE' median lag in this interval was $< 15\%$ of CliVer's in both the *basic* and *optimized* configurations. Indeed, the 25^{th} percentile of CliVer's lag in this first 30s exceeded essentially *all* lags induced by TASE in the same interval. By the end of the first 30s, both tools "caught up" and maintained lags capable of sustaining interactive use until about 90s into the traces; at this point, large server-to-client transfers caused CliVer to lag considerably in the *basic* configuration, while TASE was able to better keep up. These lags were smaller in the *optimized* configuration, since server-to-client data messages were ignored.

In Fig. 6 we report the *cost* for verifying each message in these connections as a function of the message's size. The datapoints in Fig. 6 represent all 21 TLS sessions in the Gmail dataset but, in the case of CliVer, omit points for the ClientHello message and selected handshake messages of each TLS connection. These messages were omitted because CliVer's excessive verification costs for them skewed the y-axis range considerably, rendering the other trends more difficult to distinguish visually. (All messages are included in the TASE datapoints, however.) Fig. 6a represents the costs in the *basic* configuration, and Fig. 6b shows the costs in the *optimized* configuration. As can be seen in Fig. 6b, the costs for most messages scaled linearly in message size for both TASE and CliVer, but the slope of this growth was flatter with TASE, resulting in lower costs (and so less lag) for verification. In Fig. 6a, the datapoints for TASE fell along two lines corresponding to the client-to-server and server-to-client messages (the latter are mostly omitted from Fig. 6b), and similarly for the datapoints for CliVer.



(a) *basic* configuration (b) *optimized* configuration

Fig. 6: Verification cost vs. message size

While TASE is designed to reduce the latency of symbolic execution, a secondary concern is the number of processes and amount of memory incurred by its use in behavioral verification. In Appendix A we show these statistics. Briefly, the number of processes remained roughly flat during verification, and the amount of memory used during verification grew slowly and never exceeded 3.6GB. We believe we can further reduce this memory footprint with better engineering; memory usage has not been a limiting factor for us so far.

C. Other Applications: Defending Against Memory Exploits

TASE's method of speculatively executing application code within hardware transactions and detecting when the application reads or overwrites poison values has applications for several tasks beyond full-blown symbolic execution, with minimal changes to the tool. As one example, we have prototyped a simple adaptation of TASE that places the poison value as a canary [21] adjacent to the return address in each stack frame; any application instruction that tries to overwrite the canary would then invoke the interpreter. We have further extended this design with a slightly modified memory allocator that places the poison value immediately before and after each heap-allocated buffer, providing detection of reads or writes off the beginning or end of the buffer. This modification of TASE will thus detect many common types of memory exploits, and requires only small changes to the TASE compiler to emit modified function prologues and epilogues, minor changes to the memory allocator, and small changes to the KLEE-based interpreter to diagnose an encountered error. In total, these changes comprised fewer than 140 source lines of code.

We summarize this application of TASE primarily to emphasize the flexibility of its techniques. As such, we do not quantitatively evaluate it against the plethora of available tools that provide similar properties. Qualitatively, however, this adaptation of TASE to detect memory exploits adds essentially the same latency overheads as purely concrete execution in TASE—runtimes of the same benchmarks in Table I were changed by only at most 15% when these protections were applied to *every* stack frame and heap-allocated buffer—and should impose memory overheads comparable to or less than similar tools that leverage shadow memories (e.g., [15], [42], [37], [8], [44]) or that use specialized memory allocators to achieve similar protections using page-level permissions (e.g., [3], [34], [23]). Moreover, by diverting execution to the interpreter when a memory exploit is detected, the interpreter can attempt to permit execution to continue safely (e.g., [47]), though we have not implemented these extensions presently.

We could extend this basic implementation to similarly detect use-after-free exploits, i.e., by poisoning buffers when they are freed. So, if a stale pointer were dereferenced, TASE would either trap safely with a segmentation fault if the dereferenced memory was unmapped or trigger a poison check if the memory was freed but not unmapped. Either way, the interpreter could consult its bitmap after the trap and unambiguously conclude that the pointer is no longer valid. We anticipate that this extension would induce only the runtime overhead of poisoning each buffer when it was freed and require minimal additional code modifications.

VII. Limitations

In this section we discuss several limitations of TASE.

A. Hardware Dependence on Intel TSX

To our knowledge, Intel currently provides the only widespread implementation of hardware transactional memory, limiting TASE to executing only on Intel machines with support for TSX instructions. Nevertheless, in the future, transactional memory could be implemented by other vendors;

as early as 1993, Herlihy and Moss proposed an implementation for hardware transactional memory based on "straightforward extensions to any multiprocessor cache-coherence protocol" [27].

B. Equivalence of Interpretation and Native Execution

Inevitable differences between the behaviors of native execution and interpretation of the same project imply that TASE results are not necessarily semantically equivalent to those obtained using symbolic execution based on interpretation only. For example, KLEE detects out-of-bounds accesses to concrete buffers, while native execution (without additional instrumentation) does not. It would seem that differences in the results of applying KLEE and TASE to a project could arise, however, only due to the project's processing on *concrete* values, since processing on symbolic values would trigger interpretation in TASE, as well.

In the context of the client verification application discussed in Sec. VI-B, this means that those behaviors permitted by CliVer, which uses KLEE to interpret the client program in full, are not identical to those permitted by TASE. However, a behavior permitted by TASE but not by CliVer, if caused by an input validation error of the client program, would presumably need to be an artifact of server-to-client messages, which are concrete to the verifier. Since the verifier is deployed to defend the server and is trusted to cooperate with it, malicious server-to-client messages are outside the scope of those techniques.

C. Interpreting x86 Instructions

Although the use of native state as the primary representation for program execution in TASE introduces opportunities for speculative native execution on concrete data, this design choice also introduces some difficulties.

Because KLEE requires LLVM IR to perform symbolic execution, we needed to produce LLVM IR models for the effects of each x86 instruction to be interpreted by KLEE on the program's state. In addition to providing a burdensome engineering challenge, we found (as did S^2E's authors [18]) that modeling a given x86 instruction's impact on program state using the RISC-like LLVM IR required several LLVM IR instructions to fully capture all side effects, including the changes to the FLAGS register.

As a result, a machine-independent interpretation of a source program in vanilla KLEE could require fewer LLVM IR instructions to model the program's execution than in TASE. We feel that our use cases contain a sufficiently large usage of concrete data to justify the optimizations for native execution in TASE, but a tradeoff nevertheless exists between the additional instructions needed for interpretation in TASE and the speed gained in native execution.

D. Instrumentation

In order to ensure that reads or writes to memory addresses containing symbolic values are accurately recorded, TASE uses a custom LLVM backend to emit and instrument code. Although LLVM provides many utilities for writing compiler passes to analyze or modify machine code as it is emitted, significant engineering challenges must still be overcome to ensure that all code emitted for TASE is properly instrumented. Specifically, the large number and variety of x86 instructions available, combined with their side effects and implicit operands, make it difficult to write a catch-all compiler pass that determines how an instruction touches memory. Furthermore, determining exactly where in the LLVM backend to inject instrumentation can be nontrivial, given that LLVM applies a large number of stages of optimization, some of which may modify code emitted earlier during compilation.

To simplify the instrumentation process, TASE's LLVM backend restricts the pool of x86 instructions available to the compiler during instruction selection. Our anecdotal evidence suggests that the slowdown imposed by choosing from a more limited set of instructions is negligible compared to the overhead of setting up and committing transactions and periodically interpreting when needed, but we may expand the set of allowed instructions in the future.

E. Controlled Forking

TASE was designed to use native forking to explore different execution paths, each in a different process, in order to avoid the overhead of software-based copy-on-write mechanisms as used in KLEE and S^2E [10], [18]. Although forking allows TASE to explore distinct paths in parallel, exploring distinct execution paths within distinct address spaces complicates the process of applying search heuristics across these many processes, sharing cached SMT query results across paths, etc. Furthermore, even if it were desirable to move all or some aspects of path exploration into a single address space, the TSX transactions utilized for our speculative execution scheme would likely abort more often due to their original intended use—detecting conflicting concurrent accesses—thereby impinging on performance.

As discussed in Sec. IV-E, our present implementation leverages a central manager process to guide path exploration, which it does simply by prioritizing which worker processes it allows to proceed (and temporarily suspending others). Some applications might require more sophisticated mechanisms for state management in which this simple prioritization is insufficient. For example, hybrid symbolic execution as introduced in Mayhem [13], in which symbolic states can be archived to relieve memory pressure and restored later for further exploration, might be needed for analyzing some types of applications efficiently.

VIII. Conclusion

In this paper, we presented the design, implementation, and evaluation of TASE. To our knowledge, TASE is the first symbolic execution engine that leverages specialized hardware capabilities to accelerate native execution to optimize workloads in which operations on concrete data are a major bottleneck. The two technical innovations in TASE to make this possible are (i) batching tests to detect native accesses to symbolic data into a few instructions, and (ii) undoing the potentially erroneous effects of having accessed symbolic data natively by leveraging hardware transactions.

We illustrated an application of TASE for verifying whether the messaging behavior of a client as seen by the server is consistent with the software the client is believed to be executing.

We showed that the use of TASE in this application dramatically reduced the lag associated with verifying OpenSSL TLS 1.2 traffic, e.g., as driven by Gmail. This reduction bolsters the prospects of deploying this verification on the critical path of delivering client messages to the server, as an inline defense against client exploits without foreknowledge of server vulnerabilities.

ACKNOWLEDGMENTS

We are grateful to our shepherd, Hamed Okhravi, and to the anonymous reviewers for their constructive feedback. This research was supported in part by grant N00014-17-1-2369 from the U.S. Office of Naval Research.

REFERENCES

[1] J. M. Allen, "OS and application fingerprinting techniques," https://www.sans.org/reading-room/whitepapers/tools/os-application-fingerprinting-techniques-1891, Sep. 2007.

[2] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, Mar. 2008, vol. 4963, pp. 367–381.

[3] Apple Corporation, "Enabling the malloc debugging features," https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html, 23 Apr. 2013.

[4] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, Apr. 2005, pp. 41–46.

[5] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Transactions on Information and System Security*, vol. 14, Dec. 4.

[6] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Recent Advances in Intrusion Detection, 14th International Symposium*, ser. LNCS, vol. 6961, Sep. 2011, pp. 1–20.

[7] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT – a formal system for testing and debugging programs by symbolic execution," in *International Conference on Reliable Software*, 1975, pp. 234–245.

[8] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *9th IEEE/ACM International Symposium on Code Generation and Optimization*, Apr. 2011, pp. 213–223.

[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *IEEE Symposium on Security and Privacy*, May 2006.

[10] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *ACM Conference on Computer and Communications Security*, Oct. 2006.

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[13] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[14] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjà Vu," in *12th ACM Asia Conference on Computer and Communications Security*, Apr. 2017, pp. 7–18.

[15] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *11th IEEE Symposium on Computers and Communications*, Jun. 2006.

[16] A. Chi, R. A. Cochran, M. Nesfield, M. K. Reiter, and C. Sturton, "A system to verify network behavior of known cryptographic clients," in *14th USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2017, p. 177–195.

[17] V. Chipounov, V. Kunetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems*, vol. 30, no. 1, Feb. 2012.

[18] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[19] R. A. Cochran and M. K. Reiter, "Toward online verification of client behavior in distributed applications," in *20th ISOC Network and Distributed System Security Symposium*, Feb. 2013.

[20] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: securing software by blocking bad input," in *21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[21] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *7th USENIX Security Symposium*, Jan. 1998.

[22] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "PRIME+ABORT: A timer-free high-precision L3 cache attack using Intel TSX," in *26th USENIX Security Symposium*, 2007, pp. 51–67.

[23] "D.U.M.A. – detect unintended memory access," http://duma.sourceforge.net, accessed: 25 Jul. 2020.

[24] "fork(2)," in *Linux Programmer's Manual*, 15 Sep. 2017, http://man7.org/linux/man-pages/man2/fork.2.html.

[25] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *ACM Conference on Programming Language Design and Implementation*, Jun. 2005, pp. 213–223.

[26] P. Godefroid, M. Leving, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44.

[27] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th International Symposium on Computer Architecture*, 1993, pp. 289–300.

[28] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, Oct. 2019.

[29] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[30] A. Kleen, "TSX anti patterns in lock elision code," https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code, 26 Mar. 2014.

[31] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *14th USENIX Security Symposium*, Jul. 2005, pp. 161–176.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *26th ACM Conference on Programming Language Design and Implementation*, Jun. 2005.

[33] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Transactions on Computers*, vol. 67, no. 3, Mar. 2018.

[34] Microsoft Corporation, "GFlags and PageHeap," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap, 23 May 2017.

[35] D. Milushev, W. Beck, and D. Clarke, "Noninterference via symbolic execution," in *Formal Techniques for Distributed Systems*, 2012.

[36] D. Monniaux, "A survey of satisfiability modulo theory," in *18th International Workshop on Computer Algebra in Scientific Computing*, ser. LNCS, vol. 9890, 2016, pp. 401–425.

[37] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *3rd International Conference on Virtual Execution Environments*, Jun. 2007, pp. 65–74.

[38] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *ISOC Network and Distributed System Security Symposium*, Feb. 2005.

[39] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium*, Aug. 2020, pp. 181–198.

[40] C. S. 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 *International Symposium on Software Testing and Analysis*, Jul. 2008, pp. 15–26.

[41] C. S. Păsăreanu, Q. S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-SMT," in *29th IEEE Computer Security Foundations Symposium*, 2016, pp. 387–400.

[42] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *39th IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.

[43] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *13th International Symposium on the Foundations of Software Engineering*, Sep. 2005.

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, Jun. 2012.

[45] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *ISOC Network and Distributed System Security Symposium*, Feb. 2017.

[46] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware," in *ISOC Network and Distributed System Security Symposium*, Feb. 2015.

[47] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," in *8th International Conference on Information Security*, ser. LNCS, vol. 3650, Sep. 2005, pp. 1–15.

[48] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *12th International Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 5584, 2009, pp. 244–257.

[49] N. Tillmann and J. D. Halleux, "Pex: White box test generation for .NET," in *2nd International Conference on Tests and Proofs*, 2008, pp. 134–153.

[50] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," *SIGSOFT Software Engineering Notes*, vol. 29, pp. 97–107, Jul. 2004.

[51] R. Wang, X. Wang, Z. Li, H. Tang, M. K. Reiter, and Z. Dong, "Privacy-preserving genomic computation through program specialization," in *16th ACM Conference on Computer and Communications Security*, Nov. 2009.

[52] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," in *IEEE Symposium on Security and Privacy*, May 2006.

[53] J. Yeongjin, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *ACM Conference on Computer and Communications Security*, 2016, pp. 380–392.

[54] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error diagnosis by connecting clues from run-time logs," in *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010, pp. 143–154.

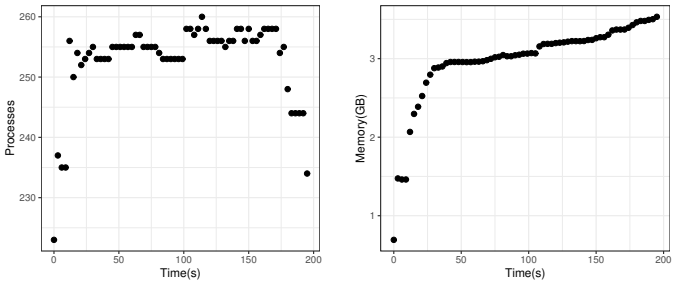
[55] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, Aug. 2018.

[56] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *5th European Conference on Computer Systems*, Apr. 2010, pp. 321–334.

[57] Z. Zhou, Z. Qian, M. K. Reiter, and Y. Zhang, "Static evaluation of noninterference using approximate model counting," in *39th IEEE Symposium on Security and Privacy*, May 2018, pp. 514–528.

APPENDIX

In this appendix, we profile the process count and memory usage of client behavioral verification using TASE. Fig. 7 shows the process counts and memory usage involved in verifying the 21 TLS sessions using the *optimized* configuration. The data reported in this figure was gathered by running the `top` command on the verification computer, with 3s snapshot intervals, while verification was being performed. Fig. 7a shows the "running" and "sleeping" processes on the platform during the verification, as a function of time. Most of the processes on the computer were unrelated to verification; i.e., over 200 processes were sleeping on the computer before verification began. However, the *growth* in the process count once verification began was due to processes involved in verification. Most importantly, however, the number of these processes stayed roughly flat after an initial spike; i.e., the rate of forking of new verification processes was roughly matched by the rate at which they exited, in this application.



(a) Running and sleeping processes

(b) Memory usage

Fig. 7: Resource usage on verifier computer during verification of 21 TLS connections (*optimized* configuration) illustrated in Fig. 4.

The memory usage on the computer grew very slowly after an initial spike and never exceeded 3.6GB; see Fig. 7b. This graph shows the benefit of leveraging copy-on-write page sharing and, in particular, sharing the binary representation of LLVM instructions to use in interpretation across all of these processes. Memory usage has not been a limiting factor for us to date, and so has not drawn our focus; we thus expect we can reduce this memory footprint further with a concerted effort to do so.